A Final Report
Grant No. N00014-90-J-1339

January 1, 1990 - December 31, 1990

SEVENTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE

Submitted to:

Scientific Officer Code: 1133
Gary M. Koob
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

Submitted by:

R. P. Cook
Associate Professor

Report No. UVA/525442/CS91/101
September 1990

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF
ENGINEERING
& APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

A Final Report
Grant No. N00014-90-J-1339

January 1, 1990 - December 31, 1990

SEVENTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE

Submitted to:

Scientific Officer Code: 1133
Gary M. Koob
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

Submitted by:

R. P. Cook
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525442/CS91/101
September 1990

Copy No. 5

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1990

3. REPORT TYPE AND DATES COVERED
Final Report 1/1/90 - 12/31/90

4. TITLE AND SUBTITLE
Seventh IEEE Workshop on Real-Time Operating Systems and Software

5. FUNDING NUMBERS
N00014-90-J-1339

6. AUTHOR(S)
Robert P. Cook

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science
School of Engineering and Applied Science
University of Virginia, Thornton Hall
Charlottesville, VA 22903-2442

8. PERFORMING ORGANIZATION REPORT NUMBER
UVA/525442/CS91/101

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Scientific Officer Code: 1133
Gary M. Koob
Office of Naval Research
800 North Quincy St., Arlington, VA 22217-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This workshop, held May 10-11, 1990 at the University of Virginia, was the seventh in a continuing series of workshops on real-time operating systems and software. This workshop, co-sponsored by the IEEE Computer Society Technical Committee on Real-Time Systems and the Office of Naval Research, had as its goals:

to investigate advances in real-time operating systems;
to promote interaction among researchers and practitioners; and
to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

This report contains the Proceedings of the Conference as well as a Final Report to IEEE and additional budgetary information provided to the Office of Naval Research.

14. SUBJECT TERMS

15. NUMBER OF PAGES
160

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
unclassified	unclassified	unclassified	unlimited

Proceedings

Seventh IEEE Workshop on Real-Time Operating Systems and Software

May 10 - 11, 1990
UNIVERSITY OF VIRGINIA
Charlottesville, VA

UNITED STATES NAVY • OFFICE OF NAVAL RESEARCH

SCHOOL OF
ENGINEERING
& APPLIED SCIENCE
at the University of Virginia

THE COMPUTER SOCIETY
OF THE IEEE

Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software

May 10 - 11, 1990
UNIVERSITY OF VIRGINIA
Charlottesville, VA
USA

Table of Contents

Operating System Support for Adaptable Real-Time Systems[†]

Thomas J. LeBlanc
Evangelos P. Markatos

Computer Science Department
University of Rochester
Rochester, New York 14627
(716) 275-9491
{leblanc,markatos}@cs.rochester.edu

Abstract

This paper outlines our plans for a real-time systems research program to support the long-term goal of developing intelligent robots. The distinguishing characteristic of our approach to real-time systems is an emphasis on system *adaptability* in a dynamic real-world environment. We achieve adaptability by allowing multiple real-time process models, with different known properties and timing constraints, to coexist within a single system and application. Using a hardware architecture in which a large-scale multiprocessor controls a behavioral system with vision and manipulation capabilities, we are constructing a prototype software environment that provides a predictable schedule for *reflexive* robot tasks and a flexible environment for implementing adaptive *cognitive* robot tasks. We will exploit multiple processors, user-level scheduling, and user-defined process and communication models in the construction of real-time robotics applications.

1. Introduction

A primary goal of modern real-time systems is predictability. Given sufficient information about each process and the state of the world, a predictable system attempts to find a schedule under which all processes meet their timing constraints. Such a schedule guarantees that timing problems will not lead to incorrect system behavior, removing a common and complex source of errors.

A real-time system can be made predictable only if the requisite information is available. This information varies with the process model and the exact form of guarantee, but generally includes attributes such as worst-case computation time, execution period, earliest starting time, criticality or priority, and synchronization constraints. A predictable system uses this information to reconcile the competing demands of the various processes in the system, creating a schedule that guarantees that all processes will meet their timing constraints. Depending on how much information is available and when it becomes available, off-line guarantees may be possible. On-line guarantees, although potentially expensive, can be used to provide both early warnings and atomic actions.

Even if complete information about every process is available, predictable systems do not make guarantees about external system behavior per se. Guarantees are only made regarding the

[†] This research was supported by NSF research grant no. CDA-8822724.

timing constraints of individual processes; the system designer must embed timing constraints on external behavior into the implementation of processes using mechanisms such as process period or priority. In some cases predictions may not even be feasible, since they are often expensive to make (many real-time scheduling problems have been proven to be computationally intractable [2]) and may require information that is unavailable (for example, the computation time for a search algorithm may be unbounded) or difficult to gather (for example, the state of the external world). In addition, a predictable schedule lasts only as long as the world remains unchanged, and therefore may be of limited value in a highly dynamic environment.

Underlying the emphasis on predictability is the assumption that ensuring correct behavior in a real-time system is primarily a problem of meeting process timing constraints. While this assumption holds true for many embedded systems, where scheduling and device management dominate computation, it does not hold for many large reactive systems, including robotics applications, which often require sophisticated software and tremendous computational power.

This paper outlines our plans for a real-time systems research program to support the long-term goal of developing intelligent robots. Using a hardware architecture in which a large-scale multiprocessor controls a behavioral system with vision and manipulation capabilities, we propose to build a software environment for real-time systems based on the Psyche multiprocessor operating system [11, 12]. The distinguishing characteristic of our approach is an emphasis on *adaptability* in a dynamic real-world environment. We plan to develop a programming environment, consisting of an operating system, library packages, and other software tools, to support the construction of adaptable real-time systems, with a particular focus on robotics applications.

Of course all systems attempt to be adaptable to some degree. The crucial distinction, in our view, between predictable and adaptable real-time systems is whether it is possible or even desirable to translate all broad timing constraints on behavior into narrow timing constraints on individual processes. For example, we view AI as a fundamental component of any adaptable system, including planning, searching, and reasoning. In such a system, timing constraints on behavioral tasks might be well specified, but an upper bound on computation time for each process might not be known. Process timing constraints may not be well-understood, or they may be so complicated that they are only approximated by very pessimistic approaches (e.g., deadlines). A predictable system would require that we limit our implementation to processes that fit a predefined, possibly complex, model.

We expect to exploit predictions when feasible; the difference is primarily one of emphasis. We are concerned with applications containing large components that do not meet the limitations imposed by predictable systems. Our intent is to explore the interface between hard and soft real-time systems, with the goal of developing mechanisms to support the construction of intelligent robots.

A representative application is a mobile robot that moves from room to room. An application program within the robot develops a plan of the form: (1) go to the door, (2) open the door, (3) enter the hallway, (4) turn right, (5) go to the next room, (6) open the door, and (7) enter. This plan must be executed within some time constraint. The application has an explicit representation of this time constraint, having formulated both the goal and the constraint. The plan is subject to dynamic modification (for example, an alternative route may be necessary if the door is locked) and may include subgoals for which no explicit time bound is given. In order to build real-time applications of this type we plan to exploit the following:

Predictable real-time systems technology - Many low-level processes in a robot implementation will have known timing constraints and computation needs. Whenever possible, we plan to exploit known techniques for static and on-line scheduling to produce guarantees.

Large-scale multiprocessors - We view the existence of multiple processors as an opportunity to add another level of abstraction to real-time applications, and not as another complication to an already intractable scheduling problem. Although we will not have nearly enough processors to assign each process a processor, we plan to separate different process models and timing constraints across different processors. This will hopefully decrease the complexity of the problem and increase the understandability of the program.

User-defined processes, synchronization, and communication - An adaptable system is difficult to construct from a limited set of tools. By allowing the user to define new process models, we can incorporate new process attributes into the scheduling algorithm. Rather than requiring the kernel to define a large set of synchronization and communication primitives for a single process model, we plan to allow the user to define both the process model and appropriate synchronization and communication primitives.

User-level scheduling - The user often has more information on which to base a scheduling decision than the kernel. Rather than widening the kernel interface to allow the user to provide all possible information, and thereby complicating the kernel scheduler, we plan to build schedulers in user space. This approach not only allows closer cooperation between the application and scheduler, it also allows multiple user-level schedulers, each tailored to a specific set of tasks.

AI planning and reasoning systems - An adaptable system must be able to evaluate its environment and choose an appropriate course of action. Rather than incorporate ad hoc solutions to specific problems, we plan to work with our AI colleagues to exploit their developments on general systems for planning and reasoning.

Many of our assumptions are shared with the designers of the Spring kernel [14], which has the goal of supporting predictable real-time AI applications. Spring uses predictable dynamic scheduling to add flexibility to systems that have historically been difficult to modify, unable to adapt to changing conditions, and based entirely on static schedules. Our emphasis is on situations where the dynamic scheduling approach employed in the Spring kernel may be infeasible or where other techniques could be fruitfully exploited, such as imprecise computations, criticality-based scheduling, and exception propagation.

Our work also shares several ideas with the CHAOS-ARC operating system kernel [4,5]. We both support hard and soft real-time constraints within a single program and allow the user to tailor scheduling policies to changing conditions. CHAOS-ARC presents a single unified computation model based on objects, invocations, and transactions, whereas we allow users to define their own process and communication models. We are also specifically interested in the use of AI techniques in real-time applications.

In what follows, we present a brief overview of the Psyche multiprocessor operating system. We then motivate our use of multiple real-time process models within a single system, describe our plans for building a real-time system based on Psyche that incorporates two distinct general models, and outline our research agenda.

2. Psyche Overview

For the past six years we have been engaged in the implementation and evaluation of systems software and applications for large-scale shared-memory multiprocessors. Based on this experience, we have become convinced that the effective implementation of a wide range of applications will require multiple models of parallelism, with differing notions of process state and scheduling, communication, sharing, and protection. Since parallelism has become so fundamental both to the programmer's conceptual model and to the effective use of the underlying hardware, existing approaches to operating system design, with a model of parallelism and communication imposed by the system kernel, are too inflexible.

We have designed and implemented the kernel of an operating system called Psyche that embodies a new approach. The principal motivation for Psyche is the support of *multi-model parallel programming*, in which application programs written under different models of parallelism can run on the same machine at the same time, and can interact in useful and well-structured ways. Psyche achieves this flexibility by presenting a user interface based on the fundamental primitives of shared-memory hardware: data, processors, subroutine calls, and protection boundaries. Rather than providing a fixed set of parallel programming abstractions, this interface provides an abstraction mechanism from which many different user-level abstractions can be built *in user space*. A fundamental assumption of this approach is that there will be a substantial amount of systems software above the kernel (in the form of library packages or programming language implementations), and below the typical user.

Psyche provides the user with three basic abstractions: the *realm*, the *process*, and the *virtual processor*. Realms form the unit of code and data sharing. Processes are user-level threads of control. Virtual processors are kernel-level abstractions of physical processors on which processes are scheduled. Processes are implemented in user space; the other two abstractions are implemented in the kernel.

The realm is a passive entity that contains code and data. The code usually consists of operations that provide a protocol for accessing the data. Since all code and data is encapsulated in realms, all computation consists of the invocation of realm operations. Interprocess communication is effected by invoking operations of realms accessible to more than one process. Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call or as safe as a heavyweight context switch. The two forms of invocation are initiated in exactly the same way, with the native architecture's jump-to-subroutine instruction. The kernel implements protected invocations by catching and interpreting page faults.

A virtual processor is the kernel-provided abstraction of a physical processor. A process must be bound to a virtual processor in order to execute. There is no fixed correspondence between virtual processors and processes; many processes will usually share a single virtual processor. A given process may run on different virtual processors at different points in time. The user is responsible for the assignment of virtual processors to physical processors. When necessary, the kernel multiplexes a single physical processor among several virtual processors using priority and round-robin scheduling.

The creation and scheduling of processes is done entirely in user-space, without kernel intervention. A data structure maintained by the user and visible to the kernel contains an indication of which process is being served by the current virtual processor. It is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space.

Communication from the kernel to virtual processors takes the form of signals that resemble software interrupts. A software interrupt occurs whenever a kernel-detected event occurs, such as a timer expiration. Using this mechanism, process schedulers can be implemented entirely in user space, managing the representation of processes and mapping them onto virtual processors.

The Psyche kernel allows the user to define the most appropriate process model and policies for resource management in each application. The average user may be unwilling or unable to fully exploit these mechanisms; therefore, we plan to implement different policies in libraries that are linked to user programs and provide convenient high-level abstractions.

Psyche is currently implemented on a 24-node BBN Butterfly Plus [1]. Our first robotics application, a balloon bouncing program called Juggler, successfully ran in November 1989. This application combines binocular camera input, a pipelined image processor, and a 6-degree-of-freedom robot arm (with a squash racquet attached) to bounce a balloon. The implementation uses a competing agent model of motor control; five processes compete with each other for access to the robot arm to position the balloon in the visual field, to position the racquet under the balloon, and to hit the balloon.

In most respects Juggler is a poor example of a real-time application. The worst possible outcome is hardly catastrophic and no individual process is time critical. The software does not need to process every input frame. Each application process is allocated a physical processor, so scheduling is not a concern. Even if processes had to share processors, failure to execute any one process during a particular time interval would have little if any affect on behavior; in the competing agent model, each application process continually broadcasts commands to the robot in competition with other processes.

Juggler does have characteristics that are representative of the type of application we plan to support, however. The only timing constraint with respect to external behavior is "hit the balloon before it hits the floor." The height of the balloon and rate of descent will vary over time. Although it is possible to translate this constraint into timing constraints on processes that filter the input, compute, and produce output commands, the result would not be the competing agent model we wished to explore.

3. Multi-Model Real-Time Systems

Just as a single, complex parallel program may incorporate several different models of parallelism and communication, a large, complex real-time system will likely have subsystems with different types of timing constraints. For example, a robot application may consist of image processing, planning, manipulation, and emergency subsystems. The image processing subsystem is periodic, executing at a rate equal to the rate of camera input. The planning subsystem controls the behavior of the robot and may utilize imprecise computations in order to find the best plan in the available time. Manipulation activities occur as dictated by the plans generated in the planning subsystem, and must meet the timing constraints given in the plan. The emergency subsystem is aperiodic; it is activated in response to asynchronous events, and must respond within narrow time constraints.

Two different approaches can be used to implement this abstract functionality in a real-time computing system. One approach is to provide a single, general process model that defines both the process attributes and timing constraints, and an appropriate scheduler. This approach requires that each process provide the necessary attribute information, such as worst-case computation time or period, and that all processes have similar timing constraints, such as deadline or earliest-start-time constraints. Efforts to expand the utility of rate monotonic scheduling [9] fall into this category. Rate monotonic scheduling assumes that processes are periodic, with known computation times, and that each process has a deadline equal to its period. This model can also support aperiodic processes with known computation times and deadlines, while still using a rate monotonic scheduler [6, 13].

The advantage of using a single process model is that all processes in the system inherit the proven properties of the scheduler, which may include on-line or off-line guarantees, and early warnings. There are several disadvantages, however. Few models have been shown to be general enough to incorporate any other models; no model is likely to incorporate the wide range of process attributes and timing constraints already present in the literature. In addition, some processes may not be able to provide the attribute information required, or may have different timing constraints than those supported by the general model. Finally, a single model that combines the attributes and timing constraints of several different types of processes may significantly complicate the development of a corresponding scheduler, and is likely to lead to an extremely pessimistic analysis.

The alternative approach, which we advocate, is to allow the system designer to define a new process model for each set of process attributes and timing constraints, and to provide a scheduler for each process model. We believe that this multi-model approach to real-time systems is preferable to the single-model approach for the following reasons:

Simplicity — A modular approach to scheduling simplifies both the development and analysis of real-time applications. For each set of processes with specialized timing constraints, we develop a process model and scheduler tailored to the attributes and constraints of that set.

Tractability — Each scheduler manages the processes of a particular model and need not worry about the processes of other models. The separation of scheduling algorithms for different process models reduces the number of constraints and attributes that must be considered, simplifying the problem for each scheduler.

Flexibility — No prespecified limit on the allowable set of attributes or constraints is imposed by the underlying system. As new combinations of process attributes and timing constraints are developed, they can be incorporated into a system by building a corresponding scheduler.

Extensibility — Each new process model receives a new scheduler. We can incorporate new process models into the system without changing existing schedulers (or the kernel, given our assumption that schedulers reside in user space).

These advantages are not without cost, however. By modularizing the scheduling problem, we may be precluded from finding a globally optimal schedule, since even a set of locally optimal solutions may not be globally optimal. We do not believe this limitation to be a serious one in practice, however, because the problem of finding a globally optimal schedule in a multiprocessor system is usually NP-hard; even single-model systems will use heuristics to find suboptimal schedules.

In addition, we have introduced a new problem: how to map the various process models and their schedulers onto hardware resources. There are two implementations we plan to explore: (1) separate process models using hardware boundaries and (2) provide each process model with a unique virtual machine, and allow virtual machines to share physical processors. The first implementation presupposes the existence of multiple processors. Hardware resources can be allocated to models statically or dynamically, but are likely to be reallocated relatively infrequently compared to the arrival and execution rate of processes. As a result, the dynamic scheduling decision does not have to balance the competing concerns of processes from different models; that balance is struck when hardware resources are allocated. It is worth noting that this result is true even if we choose the second implementation.

In the second implementation, the kernel exports to the user a virtual machine for each process model. This virtual machine must preserve the abstractions of a physical machine, including some notion of physical time. This approach is particularly appropriate for process models based on relative time (if, for example, the virtual machine slows physical time by a constant factor, but preserves relative time), but is still practical in some cases for physical time.

Finally, the existence of multiple process models within a single real-time application implies that interactions are not limited solely to shared processor resources; communication across models is likely to occur. The general problem of preserving timing constraints during communication has not been adequately addressed even when both processes involved have the same attributes and constraints, although solutions for specific cases (such as maintaining a consistent priority using a priority inversion protocol [10]) have been developed. The problem of meeting timing constraints across models is even more complex.

We believe that the advantages of multiple models outweight the disadvantages, but progress on the problems listed above is required. We have begun to explore each of these problems in the context of a specific design of a real-time system for robotics, which we describe in the next section.

4. An Adaptable Real-Time System for Robotics

Our current efforts are devoted to the development of a prototype adaptable real-time environment based on the Psyche kernel. This environment will serve as a testbed for experiments with real-time robot applications, and provide infrastructure for research on real-time systems.

As a first cut, we have divided the system into two general models, containing *reflexive* and *cognitive* processes, that emphasize predictability and adaptability, respectively. Low-level, robot life-support tasks reside in the reflexive subsystem; high-level application tasks reside in the cognitive subsystem. By dividing our hardware resources into reflexive and cognitive subsystems, we separate hard and soft real-time processes and their associated schedulers. This separation allows us to experiment with different approaches to real-time behavior using AI on a firm foundation of already developed hard-real-time technology.

The reflexive subsystem resembles a traditional hard real-time system. Processes in the reflexive subsystem are associated with I/O device management or low-level robot control. These processes are usually static in number, with bounded execution times and few (if any) failure states. This subsystem provides the mechanisms needed to implement a predictable hard-real-time system, including physical resource management, process models with priority, deadlines, or other timing constraints, and associated schedulers. For the most part, the subsystem is implemented in program libraries outside the kernel. These libraries implement resource management algorithms that give static or dynamic guarantees and early warnings.

The reflexive subsystem may consist of processes from different process models, each with their own timing constraint. Each model defines the information that is known about processes within the model, which may include the worst-case execution time, period, and deadline. For example, the processes that control the moving parts of the robot are periodic, with known period and computation times. These processes can be scheduled using a rate-monotonic scheduling algorithm. On the other hand, processes that respond to interrupts from the human controller are aperiodic, with known computation times. These processes are scheduled using the earliest-deadline-first algorithm. We could model the aperiodic processes as periodic processes and use a rate-monotonic scheduler [13] or use the modified earliest deadline first scheduler proposed in [3], which can guarantee a set of tasks at the expense of high worst-case time complexity. Rather than do so, we use cheap processors to separate the scheduling classes, simplify the scheduling algorithms, and avoid a global, pessimistic, worst-case analysis.

The cognitive subsystem is a soft real-time system. The system attempts to satisfy those timing constraints that do exist, however nothing catastrophic happens if a deadline is missed.

Most of the AI applications execute within the cognitive subsystem. Since we assume that a planner is in control of these applications and the planner establishes the relative importance of each task, processes in the cognitive subsystem have associated criticalities. A process's criticality may vary over time, depending on the relative importance of the process to the system's behavior in a dynamic environment. When no other information is available or when a global scheduling decision is required (in which case, we assume that the only known process attribute is criticality), processes in the cognitive subsystem use a criticality-based scheduler. When other information is available, such as deadline and computation time, we can use a scheduler that makes dynamic guarantees that satisfy known timing constraints.

The two subsystems will not be totally independent. Information will flow in both directions between the two subsystems. We must isolate the effects on the reflexive system of communication with processes in the cognitive system, which in general may have no known timing constraints. We must also propagate failures in the reflexive system to the cognitive system, which is capable of choosing a fallback position in the presence of hardware or software failures. The exact nature of this interface is the subject of our current work.

We are aware of the difficulties of adding real-time support to a pre-existing operating system, but we believe that our minimal kernel, which provides the ability to perform user-level scheduling and to define process models in user space, is sufficient to explore real-time issues. With user-level scheduling, we can create an optimal or suboptimal scheduler for each different process model. For example, in the case where only process deadlines are known, earliest-deadline-first provides an optimal schedule for meeting the deadlines. When other information is available, such as the number of processes and their maximum computation time, periods, earliest starting times, functions that relate the payoff of completion with time [7], the existence of imprecise computations [8], precedence relations, and synchronization and communication constraints, other scheduling algorithms can incorporate the information.

The function of the scheduler may also vary from application to application. The scheduler may be required to meet all the deadlines, meet the deadlines of the most critical processes, meet those deadlines that optimize some cost function [7], or give each process enough computation time, such that the approximation of the solution it computes is satisfactory (i.e., imprecise computations). This type of flexibility is especially important in the multiprocessor case, where optimal solutions are intractable and heuristic solutions need to be explored.

5. Research Agenda

Our research agenda is to develop the underlying principles of multi-model real-time systems, and to construct a working prototype that embodies those principles.

The primary problem we have introduced by allowing multiple real-time process models within a single application is the effect of interactions across models, both implicit (*e.g.*, shared hardware resources) and explicit (*e.g.*, communication). Our current prototype uses hardware partitions to separate the resources used by different models, but we believe software solutions are possible. We plan to explore the costs and limitations of a software solution based on multiplexed virtual machines. We also plan to determine what attributes of a process model facilitate or inhibit resource sharing.

We have just begun to investigate interfaces for communication protocols between models that use the process attributes of the respective models to ensure the timing constraints within each model. The general problem of communication across models is similar to the specific problem of priority inversion, in that the communication protocol must respect the timing constraints of the processes involved. The general problem, however, does not assume a global attribute, such as priority, that holds across all models. Therefore, we must address the problem for each pair of models, although we hope to generalize specific solutions eventually.

Our work to date on a prototype system has emphasized operating system development with an eye towards support for real-time applications. We are now in a position to begin construction of real-time Psyche. In the short term we plan to augment the kernel interface to provide the user with control over processor and memory resource allocation. Using this expanded interface, we will build the reflexive subsystem for robot control and experiment with user-level scheduling. Since naive processor assignment or scheduling can result in serious performance penalties due to spinning, extensive context switching, remote memory access, bus contention, memory contention, and communication delays, we plan to incorporate application-specific knowledge in our schedulers.

Our medium term plans for the prototype will concentrate on the cognitive subsystem. We plan to implement several process models in library code, including such concepts as synchronization and communication constraints, deadlines, criticality, and worst-case computation time. We will also explore task organization techniques, exception propagation, and multiple communication models. Using these libraries we will investigate the integration and interaction of multiple process models and incorporate AI subsystems now under development at Rochester. In doing so, we expect to develop the interface between the reflexive and cognitive subsystems.

Our long term plan is to build applications that lead to an understanding of how we might construct intelligent robots. One difficult question is an appropriate metric for success. Unlike a predictable system, the behavior of an adaptable system is expected to be probabilistic; it may be difficult to quantify alternative solutions for survival in a highly dynamic environment. Given our emphasis on the appropriate system support, one measure of success will be how easily we can integrate new behavior into an existing system. Another will be the extent to which multi-model real-time support is used in practice, and the extent to which it simplifies the construction of complex real-time systems.

References

1. BBN Advanced Computers Inc., Inside the Butterfly Plus, Oct 1987.

2. S. C. Cheng, J. A. Stankovic and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey," *IEEE Tutorial on Real-Time Systems*, 1988, pp. 150-173.

3. H. Chetto and M. Chetto, "Scheduling Periodic and Sporadic Tasks in a Real-Time System," *Information Processing Letters*, Feb 1989, pp. 177-184.

4. A. Gheith and K. Schwan, "CHAOSart: Kernel Support for Atomic Transactions in Real-Time Applications," *Proc. 19th International Symp. on Fault-Tolerant Computing*, June 1989.

5. A. Gheith and K. Schwan, "CHAOS-ARC: Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Applications," GIT-ICS-90/06, Georgia Institute of Technology, Jan 1990.

6. K. D. Gordon, L. W. Dowdy, J. Baldo and K. J. Rappoport, "Scheduling Aperiodic Tasks with Hard Deadlines in a Rate Monotonic Framework," *Proc. 6th IEEE Workshop on Real-time Operating Systems and Software*, Pittsburgh, PA, May 1989, pp. 1-5.

7. E. D. Jensen, C. D. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proc. 6th Real-Time Systems Symp.*, San Diego, CA, Dec 1985, pp. 112-122.

8. K. J. Lin, S. Natarajan and J. W. S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *Proc. 8th Real-Time Systems Symp.*, 1987, pp. 210-217.

9. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM 20*, 1 (Jan 1973), pp. 46-61.

10. R. Rajkumar, L. Sha and J. P. Lehocsky, "An Experimental Investigation of Synchronization Protocols," *Proc. 6th IEEE Workshop on Real-time Operating Systems and Software*, Pittsburgh, PA, May 1989, pp. 11-17.

11. M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proc. 1988 International Conference on Parallel Processing*, St. Charles, IL, Aug 1988, pp. 255-262.

12. M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," TR 309, Department of Computer Science, University of Rochester, Mar 1989.

13. L. Sha, J. Lehoczky and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc. 7th Real-Time Systems Symp.*, New Orleans, LA, 1986, pp. 181-191.

14. J. Stankovic and K. Ramamritham, "The Design of the Spring Kernel," *Proc. 4th IEEE Workshop on Real-Time Operating Systems*, 1987, pp. 19-23.

CHAOS^{arc} : A Kernel for Predictable Programs in Dynamic Real-time Systems

Karsten Schwan
Ahmed Gheith

School of Information and Computer Science
Georgia Institute of Technology
ATlanta, GA 30332

January 30, 1990

1 Introduction and Review of Previous Results

The control software of real-time systems cannot be termed *reliable* unless it exhibits two key attributes:

1. Computations must complete within well-defined timing constraints typically captured by execution *deadlines*[17].
2. Programs must exhibit predictable behavior in the presence of uncertain operating environments.

We posit that (2) requires that any real-time operating system kernel must provide constructs that may be used to *guarantee* desired performance and functionality levels of selected computations in real-time applications[6]. The CHAOS^{arc} [1] object-based operating system kernel provides constructs that deal with uncertainty by allowing programs to be *adaptable*[27] (i.e., changeable at runtime) in performance[20] and functionality to varying operating conditions. Adaptations may anticipate changes in the operating environment – termed *preventive adaptations* – or they may be reactions to external events – termed *reactive adaptations.*

Our past research regarding real-time systems has dealt with preventive adaptation algorithms[20, 3] and with programming and operating system support for high-performance, adaptable control software[21, 22, 28, 25, 29, 9] – termed *operating software* – using embedded multiprocessors and a complex real-time application (the ASV autonomous suspension vehicle[18]). Preventive adaptations may *attempt* to guarantee certain levels of performance or functionality in operating software by adjusting system behavior or parameters by prediction of future from past system behavior[3, 9, 11].

The CHAOS^{arc} kernel as described here primarily targets *reactive* adaptations. However, we are now building programming and monitoring system support for CHAOS^{arc} based on the facilities described in [3, 11, 29] so that such adaptations may be made easily for programs running with the CHAOS^{arc} kernel.

2 Summary of CHAOS^{arc} Mechanisms

Reactive adaptations typically imply dynamic changes in software in response to unexpected external events, like failures, temporary overloads, etc. Thus, CHAOS^{arc} must be capable of performing exception and event handling, synchronization, scheduling, and atomic computations[32]. However, CHAOS^{arc} differs from related research in distributed databases and network operating systems in several ways. First, as with the CHAOS operating system[8, 25], CHAOS^{arc} 's object model is tailored to real-time systems. Namely, object invocations may be sporadic, time-driven (periodic), or event-driven, and they may have real-time attributes such as delays, deadlines, and criticalness. Deadline semantics may vary

[1] A Concurrent, Hierarchical, Adaptable Operating System supporting atomic, real-time computations.

from *guaranteed deadlines*, which are hard deadlines[17] that must not be missed, to *soft deadlines* that may be missed occasionally, to *recoverable deadlines* that cause programmed recovery actions when missed, to *weak deadlines*, which specify that partial or incomplete results are acceptable when the deadline is missed.

Invocations are executed by multi-grain tasks ranging from procedures executed synchronously in the caller's address space, to single or multiple execution threads[1, 19, 21, 22], which may be executed asynchronously and in parallel with the invoking task. Multi-grain tasks are complemented by multi-grain invocations, which range from reliable invocations that maintain parameters and return information (or even communication 'streams'[25]) to invocations that implement unreliable 'control signals' or 'pulses'. Thus, the first contribution of the CHAOS^{arc} kernel is its support for the assembly of efficient representations of multi-grain objects and multi-type invocations, thereby providing significant flexibility in the available operating system constructs compared to other systems[12, 2].

The second contribution of CHAOS^{arc} is its facility for guaranteeing the execution of selected parts of real-time programs, using the notion of real-time, atomic computations – for simplicity, henceforth termed 'atomic computations' – consisting of sets of atomic, real-time invocations. Each set defines a grouping of related invocations that is to be viewed and guaranteed as a single execution unit. For each atomic computation in CHAOS^{arc} , three different classes of attributes are maintained: real-time, concurrency control, and recovery attributes. *Real-time attributes* specify temporal restrictions on computation execution. *Concurrency control attributes* are constraints regarding the execution of concurrent atomic computations caused by the sharing of resources, such as serializability. Finally, *recovery attributes* are application-dependent properties required to guarantee that an incomplete (aborted) atomic computation leaves the system in a state semantically equivalent to the state before its execution.

The third difference of CHAOS^{arc} to other research in operating systems is its support for dynamic concurrency control and task scheduling policies, due to the potentially unpredictable operating conditions of real-time applications. These policies are novel in that they must themselves be *predictable* which implies that arbitrary delays due to locking in concurrency control protocols or due to cascaded aborts in timestamping-based protocols are not acceptable.

The primary motivation for the development of CHAOS^{arc} are our experiences with several, highly complex real-time applications, which were or are being constructed in conjunction with researchers in academic and industrial environments[15]. In that work, it has become clear that the programming of such applications requires high-level, efficient programming and operating system mechanisms, ranging from notions of parallel tasks and task communication[21, 22], to objects and object invocations[25], to finally, atomic computations[6]. Parallelism is required because most real-time systems' operating environments exhibit substantial concurrency. Dynamic support for objects, which potentially exhibit internal parallelism, allows programmers to develop and operate with more complex abstractions than those programmable with tasks, yet not sacrifice efficiency due to alterations of the basic object model implemented by CHAOS[25]. Finally, atomic computations permit programmers to state and maintain statically determined global properties of their multi-object applications[10], thus enabling them to cope better with dynamic, unpredictable variations in operating environments.

In the remainder of this paper, we concentrate on the object-based kernel of CHAOS^{arc} , in order to demonstrate the implementation of CHAOS^{arc} objects. A brief description of an implementation on a BBN Butterfly Plus multiprocessor follows. The concurrency control protocols for atomic computations and the dynamic scheduling of atomic object invocations are not presented here.

3 CHAOS^{arc} Kernel Implementation

A previous publication has demonstrated the importance of real-time, atomic computations and invocations, using examples from several real-time applications[6]. It also demonstrated that the appropriate implementation of both require efficient lower level mechanisms for representation of multi-grain objects, for synchronization, and for object invocation. This section describes the fashion in which CHAOS^{arc} 's operating system mechanisms support (1) non-atomic object invocations and the association of recovery actions with objects and invocations and (2) the association of schedulers and concurrency control algorithms with objects (e.g., to guarantee certain execution deadlines), resulting in atomic invocations and computations.

Two major aims of the design and implementation of the CHAOS^{arc} mechanisms are *predictability* and *efficiency*. Predictability means that kernel mechanisms and policies have "well behaved timing properties", that is, there should be known upper bounds on the execution times of all kernel functions. Also, the kernel must be *accountable*[8], which means that it must either honor its critical commitments or report its failure to do so to higher software levels before such knowledge becomes obsolete (provided that the underlying hardware remains in application-specific "safe states"). For example, an unanticipated change in the system noted by the kernel might cause an invocation to miss its hard deadline. This fact should be reported to the invoker within a time bound that enables the application to react accordingly. *Efficiency* requires that the guaranteed upper bounds on the execution times of the kernel mechanisms be as tight as possible.

The requirements of predictability and efficiency result in the following restrictions on the mechanisms of the CHAOS^{arc} kernel. First, limits are imposed on the complexity of the decision-making policies used within the kernel. Second, mechanisms that provide statistical performance improvements are not used, since an accountable kernel must schedule activities based on their worst case performance. However, it is possible to use predictable variations of such techniques, such as an implementation of caching by making local copies of globally accessible data. Third, resource sharing between different kernel components must be organized such that the performance of one component depends on the resource but not on other components accessing the resource.

3.1 Objects

A typical single or multi-thread object in CHAOS^{arc} has three components: shared and exclusive *state*, the *scheduler*, and *servers*. The object's *shared state* is accessible to all invocations of the same object. It is the task of the object's concurrency control and failure recovery to ascertain that multiple invocations have a consistent view of the shared state. In contrast, copies of the object's *exclusive state* are used with each atomic invocation, and this state is updated only upon invocation commit. Thus, the exclusive state facilitates backward recovery, while the shared state will be used for forward recovery[31].

The object *scheduler* receives and schedules all invocations to the object. It employs some scheduling policy to assign invocations to execution threads based on scheduling attributes (e.g., deadline) and in cooperation with processor schedulers (e.g., to attempt to guarantee deadlines[23]). All invocations to a single thread object are scheduled and executed serially by its thread. Invocations to a multi-thread object are scheduled to be executed by one of its large number of threads. Thus, multi-thread scheduling typically consists of both the assignment of threads to processors[24] and the determination of an exact schedule for the assigned thread on its target processor[33, 23]. Each scheduling decision is checked for legality by the object's *concurrency control* algorithm, which must ensure object consistency when invoked by concurrent, atomic invocations. The *commit/abort protocol* of the object ensures that a committed atomic computation has a permanent effect on the object, whereas the abort of an atomic computation leaves either no changes (backward recovery) or results in the attainment of some equivalent state regarding this object (forward recovery).

Server queues describe all available servers (i.e.,threads), each of which is generic in that it may execute any of the object's operations (in contrast to CHAOS[25]) and has its own execution environment and copy of the object's exclusive state.

Object representation. Three levels of abstraction exist in the CHAOS^{arc} system. At the first and lowest level, CHAOS^{arc} hides most of the details of the target architecture using a user-level multiprocessor threads package[19] compatible with Mach cthreads [2]. CHAOS^{arc} can execute on any machine on which such a threads package exists, which currently are a Sequent Symmetry, a BBN Butterfly, and a non-shared memory multiprocessor consisting of Intel 386 and custom processors. Two extensions of the threads package are underway. First, the package is being extended to include real-time scheduling algorithms for use by CHAOS^{arc} [23]. Second, the package's lack of predictability (mostly due to memory management in the underlying Mach operating system) and efficiency are being improved by its re-implementation on the bare hardware of the BBN Butterfly.

[2]All threads created by a single user process share the process' address space yet have their own execution states.

The second level of abstraction in the CHAOS^{arc} kernel implements *primitive* objects. This layer of software is not visible to typical applications programmers. It provides the notions of *classes*, *objects*, *invocations*, and *attributes*. Attributes can be specified for classes, objects, or invocations. The primitive layer does not associate semantics with attributes; instead, they are passed to special objects called *policies* which interpret them and implement their semantics.

At the third level, complex classes, objects, and invocations are implemented by associating a policy that implements the CHAOS^{arc} semantics (called 'ca_policy') with primitive objects. It is at this level that application programmers declare, instantiate and invoke objects for use in real-time applications. Furthermore, non-real-time complex objects may be constructed at the third level, if desired. For example, the object and invocation semantics of Presto[2], Clouds[5], HPC and its derivatives[13], or remote procedure calls may be implemented at this level.

Below, we briefly touch upon primitive objects and the manner in which complex objects are constructed from them. This discussion provides a basis for the explanation of how CHAOS^{arc} 's different types of invocation primitives are implemented.

Primitive classes and object creation. ¿From the kernel's point of view, each object is of some well-defined *class*, which defines the internal state, the operations, and the components of the object. The kernel provides constructs to create four different *primitive classes*[3] – termed *ADT*, *TADT*, *monitor*, and *task* (explained below), each of which may be created with an arbitrary number of components, thereby resulting in a component hierarchy[26].

The creation of a class makes its code, internal data structures, and class descriptors available on the target machine[26], so that instances of objects using such code and data may be created with:

```
objid = object_create (class, name, node, state)
```

This call creates an object pointed to by 'objid' of 'class' with string name 'name' on processor 'node'. 'State' is an optional pointer to an already allocated area of memory to be used as the object's state (a null value 'state' results in dynamic allocation).

A primitive object of class *ADT* (abstract data type) is passive[25], has no components (unless explicitly specified), and has a well-defined internal state and operations, which are executed in the address space and by the execution thread of the invoker (caller). However, on the BBN Butterfly multiprocessor, classes and object instances are created such that the invocation of an ADT object results in the access to a local copy of the object's code and read-only data. Thus, ADT objects may be used to implement predictable, small grain activities that are not performed in parallel with the invoker, such as an access to a data repository shared by some objects. Note that unlike the Clouds[5] system and for reasons of predictability and efficiency, such an access does not involve migration of the execution thread from the processor at which the call originates to the processor on which the target ADT resides.

An object of type *monitor* is a passive object that allows exactly one execution thread at a time to execute its operations. Monitor objects behave like Hoare monitors, with the exception that their explicitly specified scheduling policies (for the selection of invocations to be executed) may differ among instances.

An object of class *TADT* (threaded abstract data type) is active and is used for the representation of parallelism in CHAOS^{arc} applications. A TADT object creates and starts a new execution thread for the execution of each operation invocation (thread creation is very fast in the threads package used at the lowest level of CHAOS^{arc}). In contrast to the CHAOS system[25] and for reasons of efficiency and predictability at the second level of CHAOS^{arc} , all such threads are executed on the processor on which the object instance of class TADT has been created.

A CHAOS^{arc} *task* object is like an Ada task in that it consists of a single active thread of control and has multiple entry points selected by this thread. However, unlike Ada, a CHAOS^{arc} task object may use arbitrary policies for entry selection.

A complex CHAOS^{arc} object with internal parallelism is constructed from multiple instances of TADTs. The complex object's scheduler that selects one of the object's target TADTs for the execution of an invocation may be represented with an ADT or *monitor* object (resulting in the execution of the scheduler in the address space of the invoker) or with a TADT or *task* object (resulting in the execution of the scheduling code by a thread potentially executing concurrently with the invoker). CHAOS^{arc} makes this

[3]The term *primitive object* is used to denote an object of a certain primitive class.

possible by automatically re-directing each complex object's invocation to the ADT, monitor, TADT, or task acting as its *policy*. Such policies queue and schedule invocations, check concurrency constraints, and finally invoke the desired operation of the target complex object [4].

Typically, either a monitor or a task object are used for policy implementation. This guarantees that all scheduling decisions regarding the invocations of a single object are serialized.

Once constructed, primitive objects are invoked using the call:

```
object_call (object, operation, arguments)
```

This call is mapped (at compile time, if possible) into one of 'ADT_call', 'TADT_call', 'monitor_call', or 'task_call' depending on the class of the 'object' parameter.

Complex objects. Neither threads nor primitive classes are visible to the typical application programmer. Instead, users employ *complex objects* with classes that are composites of primitive classes. The complex objects currently available at the third level of CHAOS^{arc} are (1) *passive objects*, which are essentially ADTs (with or without component ADTs), are executed in the execution thread of the invoker, and are invoked via procedure-call like 'object_call' operations, and (2) *active objects*, which have their own execution threads and may be executed concurrently with the invoker. For performance reasons, atomic invocations of passive objects are not currently available, whereas active objects may be invoked atomically or non-atomically with one of six different types of invocations.

Additional object classes and invocation types are easily added to the complex level of CHAOS^{arc} , by composing them using the primitive classes in its lower level. To illustrate the manner in which such compositions of objects and invocation types are made in CHAOS^{arc} , we next describe the composition of a 'typical' complex object composed of CHAOS^{arc} 's primitive objects:

- The user interface of any CHAOS^{arc} object is its 'ca_policy', which is an ADT to which all invocations of the CHAOS^{arc} object are routed. Specifically, the 'ca_policy' ADT accepts invocations with attributes specific to CHAOS^{arc} (e.g., *atomic*, *stream*, ...), interprets those attributes, performs any necessary resource analysis and scheduling, and then initiates the execution of the body of code implementing the invocation. For resource management, the 'ca_policy' may interact with other components of the complex object or of the system (e.g., by interaction with other policies of other objects). In addition, the 'ca_policy' ADT records (as part of its state) information about the object it is managing. Examples of such information are the locking requirements of different operations (which are statically specified in the class definition) and information about currently active/pending invocations (which is dynamically manipulated). Furthermore, the 'ca_policy' ADT implementing the complex object has three component objects, which are described next.

- First, the 'resource_manager' component of the 'ca_policy' ADT is a *task* object that maintains and analyses the resource requirements of invocations. The resources present in CHAOS^{arc} are *processors* and *locks*. Regarding processors, the resource manager interacts with the lower level scheduling policy (at the thread level) to perform schedulability analysis and processor allocation for each incoming invocation. For locks, the resource manager maintains information that is used by the concurrency control policy to accept, reject, or abort individual invocations.

- Second, the 'servers' component of the 'ca_policy' contains the code for the complex object's invocations. Using this component, the 'ca_policy' creates the threads for code execution and performs the invocation to thread mapping. Such threads can be either dynamically created at invocation time or statically created at the time of object initialization.

- Third, the 'pool' component of the 'ca_policy' is used for implementation of queues for pending invocations.

Note that the apparent complexity of complex object composition is not visible to the application programmer. First, at runtime all required invocation parameters like deadlines and start times are simply stated as attributes to the complex object's invocations (the COLD^{arc} language is described elsewhere[6]).

[4]Policies are also used for implementation of the different invocation primitives available in CHAOS^{arc} .

Invisibly to the programmer, such attributes are automatically passed to the object's 'ca_policy', which in turns uses them for invocations of the its component objects. For example, the 'resource manager' component uses such parameters to schedule each invocation based on its internal concurrency control, commit/abort, and scheduling information (represented in its internal data structures). Second, the construction of the complex object's structure from primitive objects is performed by declaration of a CHAOS^{arc} object with some pre-defined class, which results in the use of a suitable 'ca_policy' object for invocation processing. Third, in [7] we showed that the apparent complexity of CHAOS^{arc} 's complex objects does not adversely affect system performance or predictability of execution. Next, we show the manner in which different types of invocations may be composed using the CHAOS^{arc} mechanisms.

3.2 Object Invocations

A large variety of invocation primitives is required by typical real-time applications. In fact, and as already shown by our earlier experiences with the CHAOS operating system[25], it is even desirable to allow users to synthesize new primitives from basic building blocks[11] offered by the real-time kernel. CHAOS^{arc} adopts the building block approach, but offers additional functionality compared to the CHAOS operating system. Specifically, in CHAOS^{arc} , users can construct arbitrary invocation primitives by use of existing building blocks as well as by creation of their own building blocks.

To illustrate, the implementation of alternate invocation types for a single target object is described next (using the complex object in the previous section). Recall that all invocations of the complex object were re-directed to the 'ca_policy' object that performed scheduling and concurrency control of invocations. In order to accommodate the different invocation types offered for CHAOS^{arc} 's complex objects, the 'ca_policy' ADT also provides a number of 'operations' (entry points). Each invocation type is implemented by one of these operations; invocations of a CHAOS^{arc} object are re-directed to the appropriate operation of the 'ca_policy' object. For example, consider an invocation of 'object$operation of a complex object:

```
INVOKE object$operation (parameters) INVID: inv[i]
    top-level, atomic,
    feedback :  success+failure;
```

For this invocation, the specific operation of the 'ca_policy' object to be invoked is selected using the default (not explicitly stated) *attribute* 'regular' of the invocation. The resulting stub for the regular invocation of 'object$operation' actually performs an invocation of the operation 'regular' of the 'ca_policy' object associated with the complex object. This invocation uses an *invocation block* (parameter block from the point of view of the policy object) containing the fields: 'object' and 'operation' concerning the complex object being invoked, 'arguments' pointing to a parameter block containing the values of the parameters being passed to 'object$operation'. 'Regular', 'top-level', and 'atomic' are *attributes* of the invocation; they are passed a arguments to the policy object. Additional parameters, such as deadlines, are also specified as attributes of the invocation.

The implementation of the object layer of CHAOS^{arc} pays detailed attention to achieving predictability and efficiency, in each of the different stages of each invocation's execution[25]: name binding, queueing, resource allocation, and termination. For example, regarding name binding, for each invocation, the specified object and operation name are statically mapped to a pointer to an ADT acting as a *address block* for the invoked object. In CHAOS^{arc} , this ADT's code is executed by the invoker and its execution results in (1) the selection of some queue for posting the invocation request and (2) request queueing. Predictability in (1) is achieved by disallowing dynamic binding and maintaining with each object a list of pointers to all address blocks of the objects it invokes. This list is indexed by object identifiers that are assigned by the language processor and are resolved when the object is initialized. As a result, once this table is initialized, name binding may be performed in constant time. Similar implementation decisions are made for queueing, resource allocation, execution, and termination of invocations.

4 Conclusions and Related Research

CHAOS^{arc} demonstrates the soundness of the notion of atomic computations in programming complex, real-time, embedded applications. Specifically, atomic computations provide a natural paradigm for grouping multiple activities into related units (computations) with guaranteed properties (e.g., deadlines, mutual exclusion, etc.). The mechanisms implemented by the CHAOS^{arc} kernel provide a predictable, accountable, and efficient basis for programming with real-time atomic computations. These mechanisms are predictable because they have well defined upper bounds on their execution times that are (can be) determined before their execution. They are accountable because their decisions are guaranteed to be honored as long as the system is in an application-specific "safe state". Finally, they are efficient because of their low execution overhead, concurrency exploitation, and tightness of their guaranteed upper bounds.

The kernel mechanisms of CHAOS^{arc} may be used to compose different classes of objects[4, 2], as well as different types of object invocations, at small performance penalties. This makes the *synthesis* of object invocations possible[11].

CHAOS^{arc} is an extension of CHAOS[25], offering atomic real-time computations as its major new concept. However, CHAOS^{arc} 's implementation and implementation concepts (e.g., classes and their use in the construction of complex objects and new types of invocations) are entirely different, which leads to some interesting properties of the CHAOS^{arc} system, such as its increased predictability compared to CHAOS and its ability to support a much larger variety of invocation types and complex object classes than CHAOS[8, 25]. In addition, CHAOS^{arc} is portable to any machine offering a Mach-compatible threads package[19] (this currently includes SUN 3 and 386 workstations, the Sequent Symmetry, and the Encore Multimax).

The fashion in which the CHAOS^{arc} kernel allows types of objects and invocations to be composed from primitive objects offers substantially increased flexibility of operating system primitives compared to other recent work in RPC[12]. Similar issues are also being explored with the PRESTO system[2] (PRESTO was developed after CHAOS[25] and concurrently with CHAOS^{arc}). The Synthesis kernel addresses adaptability at a lower level than CHAOS^{arc} [14]. It is concerned with adaptations of code segments used for the implementation of objects, whereas CHAOS^{arc} adapts object and invocation implementations at the level of primitive objects. The Spring kernel[30] differs from our work in that its primary execution environment is a distributed system.

The SARTOR project[16] supports the partially automatic, static synthesis of time-constrained program components, in more generality than the environment extensions of CHAOS (and now CHAOS^{arc}).

The future research of our group concerns the continued development of (1) the predictable CHAOS^{arc} kernel, (2) the CHAOS^{arc} programming system based on the COLD^{arc} real-time programming langage, and (3) dynamic scheduling algorithms and models.

Acknowledgements. Yiannis Samiotakis implemented the cthreads package on the Butterfly Plus as part of his M.Sc. thesis at The Ohio State University. A robotics application running on the CHAOS^{arc} kernel was written in part by Peter Wiley (also at Ohio State as part of his M.Sc. thesis), and Prabha Gopinath and Harold Forbes have implemented additional real-time programs now being used with CHAOS^{arc} . Hongyi Zhou is designing and implementing a real-time threads package for CHAOS^{arc} and David Rosenbaum is implementing the bare hardware version of our cthreads package on the BBN Butterfly Plus.

References

[1] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, May 1986.

[2] Brian Bershad, Edward Lazowska, Heny Levy, and David Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems*. ACM SIGPLAN, Jul. 1988.

[3] T. Bihari and K. Schwan. A comparison of four adaptation algorithms for increasing the reliability of real-time software. In *Ninth Real-Time Systems Symposium, Huntsville, AL*, Dec. 1988.

[4] G.A. Curry and R.M. Ayers. Experience with traits in the xerox star workstation. *IEEE Transactions on Software Engineering*, SE-10(5):519-527, Sept. 1984.

[5] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. *The 9th. International Conference on Distributed Computing Systems, San José, CA.*, June 1988.

[6] Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing, Chicago, ILL*, pages 462–469, June 1989.

[7] Ahmed Gheith and Karsten Schwan. CHAOS-ARC : Kernel support for multi-weight objects, invocations, and atomicity in real-time applications. Technical Report GIT-ICS-90/06, Georgia Institute of Technology, Jan 1990.

[8] Prabha Gopinath. *Programming and Execution of Object-Based, Parallel, Hard Real-Time Applications.* PhD thesis, Department of Computer and Information Sciences, The Ohio State University, June 1988.

[9] Prabha Gopinath, Tom Bihari, and Karsten Schwan. Object-oriented design of real-time software. In *10th International Real-time Systems Symposium, Los Angeles*, pages 194-201. IEEE, Dec. 1989.

[10] Prabha Gopinath, Tom Bihari, Karsten Schwan, and Ahmed Gheith. Operating system constructs for managing real-time software complexity. In *Proceedings of 1989 Workshop on Operating Systems for Mission Critical Computing, ONR, Maryland*, Sept. 1989.

[11] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106-125, July 1989.

[12] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the x-Kernel: Evaluating new design techniques. In *Proceedings of the 12th Symposium on Operating Systems Principles.* Assoc. Comput. Mach., Dec. 1989.

[13] Thomas J. LeBlanc and S.A. Friedberg. Hierarchical process composition in distributed operating systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado*, pages 26-34. IEEE, ACM, May 1985.

[14] Henry Massalin and Calton Pu. Threads and input / output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles.* Assoc. Comput. Mach., Dec. 1989.

[15] R.B. McGhee, D.E. Orin, D.R. Pugh, and M.R. Patterson. A hierarchically-structured system for computer control of a hexapod walking machine. In *Proceedings of 5th IFTOMM Symposium on Robots and Manipulator Systems, Udine, Italy.* IFTOMM, June 1984.

[16] Aloysius K. Mok. SARTOR - a design environment for real-time systems. In *Proceedings of 9th COMPSAC Computer Software and Applications Conference.* IEEE, October 1985.

[17] Aloysius Ka-Lau Mok. *Fundamental Problems of Distributed Systems for the Hard Real- Time Environment.* PhD thesis, Laboratory for Computer Science, Massachussetts Institute of Technology, May 1983.

[18] David E. Orin. Supervisory control of a multilegged robot. *International Journal of Robotic Research*, 1(1), Spring 1982.

[19] Yiannis Samiotakis and Karsten Schwan. A thread library for the BBN butterfly multiprocessor. Technical Report OSU-CISRC-6/88-TR19, Computer and Information Science Department, The Ohio State University, June 1988.

[20] Karsten Schwan, Thomas E. Bihari, and Ben Blake. Adaptive, reliable software for distributed and parallel, real-time systems. In *Sixth Symposium on Reliability in Distributed Software, Williamsburg, Virginia*, pages 32–44. IEEE, March 1987.

[21] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. Gem: Operating system primitives for robots and real-time control. In *Proceedings of the International Conference on Robotics and Automation, St. Louis, Missouri*, pages 807–813. IEEE, March 1985. Extended abstract. Also published as article in ACM TOCS, and available as technical report OSU-CISRC-86TR1KS.

[22] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[23] Karsten Schwan and Ben Blake. A fast scheduling mechanism for real-time systems. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-5/87-TR16, Sept. 1987. Submitted for publication in IEEE TSE since May 1987.

[24] Karsten Schwan and Cheryl Gaimon. Automating resource allocation for multiprocessors. *Journal of Systems and Software*, pages 1–16, Dec. 1988.

[25] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[26] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385–401, March 1986.

[27] Karsten Schwan and Rajiv Ramnath. Adaptable operating software for manufacturing systems and robots: A computer science research agenda. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, pages 255–262. IEEE, Dec. 1984.

[28] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A system for parallel programming. In *9th International Conference on Software Engineering, Monterey, CA*, pages 270–282. IEEE, ACM, March 1987. Awarded best paper.

[29] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A language and system for parallel programming. *IEEE Transactions on Software Engineering*, April 1988.

[30] John A. Stankovic. The design of the spring kernel. In *IEEE Real-time Systems Symposium*, pages 146–157, 1987.

[31] Hideyuki Tokuda. Compensatable atomic objects in object-oriented operating systems. In *Proc. Pacific Computer Communication Symposium*, October 1985.

[32] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[33] W. Zhao, K. Ramamritham, and J.A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.

Implementing a Predictable Real-Time Multiprocessor Kernel – The Spring Kernel.

L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

April 12, 1990

1. Introduction

The Spring paradigm [6] advocates *predictable* [5] real-time computing. The purpose of predictable real-time computing is to allow the timing properties of both individual tasks and the overall system to be assessed. The construction of predictable systems can be viewed from the bottom up – predictable architectural features facilitate the construction of predictable OS software, which leads to building predictable real-time application software.

Among the architectural requirements for predictable real-time systems are bounded instruction execution and memory access times, and bounded inter-process and inter-node communication costs. These architectural features facilitate the construction of predictable OS features such as bounded dispatching, scheduling and synchronization costs, bounded OS primitives, and bounded code execution times.

The design and implementation of the Spring real-time multiprocessor kernel supports these features. In this short paper we describe the Spring kernel support for bounded dispatching, scheduling and synchronization.

2. Overview of the Spring System

The Spring system [6] is physically distributed and is composed of a network of multiprocessors. Each multiprocessor contains one or more application processors, one or more system processors, and an I/O subsystem. System processors offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and so that this overhead does not cause uncertainty in executing guaranteed tasks. All system tasks are resident in the memory of the system processors. The I/O subsystem is a separate entity from the Spring kernel and it handles non-critical I/O, slow I/O devices, and fast sensors.

Version 1 of the Spring kernel concentrates on the multiprocessor aspect of the Spring system.

This work is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332.

A Spring node is a multiprocessor consisting of up to five [1] Motorola 68020 based MVME136A boards. The MVME136A boards support features which are typical of shared bus multiprocessors – an asynchronous bus interface, architectural support for *test-and-set* like operations, and a local memory. This memory can either be accessed remotely over the VME bus by (typically) another processor, or locally by the processor which has mapped this local memory. Additional support for multiprocessing is provided through the use of the MPCSR (MultiProcessor Control/Status Registers). The MPCSR provides the ability to generate interrupts on a selected board, and/or a simultaneous interrupt to multiple boards.

One node consists of a system board (which executes the scheduler) and multiple application boards. The dispatchers, one per application board, are responsible for the dispatching of application tasks. The scheduler and dispatcher processes are thus designed to run in parallel. External events represent invocations of application tasks with arrival times, deadlines, resource requirements, and other attributes. When a task arrives, the scheduler attempts to dynamically guarantee that the new task will meet its deadline. As tasks are guaranteed, the scheduler adds them to a system task table (STT); these tasks are also linked into dispatcher queues. Since the STT resides on the system board, a dispatch queue reference performed by the dispatcher accesses the shared bus.

Tasks are classified into three categories – critical, essential, and non-essential [6]. The online guarantee is used for *essential* tasks. These tasks have deadlines and are important to the operation of the system, but will not cause a catastrophe if they are not finished on time. It is necessary to treat such tasks in a dynamic manner as it is impossible to reserve enough resources for all contingencies with respect to these tasks.

The memory model underlying the Spring kernel design is a *local* memory model. Each processor is equipped with a local memory module; every processor can also access all other memory modules via a common bus. This models multiprocessor systems in which each processor has local memory for task code and private resources, while at the same time there are other resources, such as shared data structures, files, and communication ports, which can be used by tasks residing on different processors. This model does in fact match the 68020 based multiprocessor architecture that Spring runs on. Since each processor has its own local memory, the assignment of tasks to processors, done statically, determines which processor's memory the task code is resident. To avoid unpredictable blocking of tasks due to resource contention at run time, our scheduling algorithm integrates tasks' timing constraints and their resource requirements [4].

[1] Although eight slots exist on the backplane, only five boards can be used because of power supply limitations.

3. Foundations of the Spring OS: Scheduler and Dispatchers

Predictability of the underlying real-time OS is necessary to achieve predictability of software (application tasks) running on top of this OS. This section describes the design and implementation of significant components of the multiprocessor real-time OS – the *scheduler* and the *dispatchers*. To ensure predictability of application tasks, both the scheduler cost and the dispatching costs must be bounded. Version 1 of Spring supports a scheduler which executes in time O(N) [4] where N is the number of tasks at the node. However, the execution time of the scheduler is capped to a fixed worst case time. This will be discussed further in section 3.1. The dispatching cost is bounded by a constant. Multiple dispatchers operate concurrently with no inter-dispatcher interference. Dispatchers and the scheduler require concurrent access to the STT. Correctness of this access is maintained via the use of critical sections, while predictability is ensured by constructing all critical sections to execute in constant time.

The STT is a key data structure in the Spring kernel. Tasks which have been guaranteed are placed in the STT by the scheduler. The STT, residing in the system board memory, contains dynamic task (invocation) information and information for OS management and scheduling. The OS management and scheduling fields include fields for maintaining scheduler data structures, as well as fields for constructing linked lists which order the STT.

Concurrent execution of dispatchers is achieved by partitioning the STT based on the processor to which tasks are assigned. Each per-processor partition of the STT is known as the dispatch queue. Since a task is assigned to exactly one processor, the multiple dispatcher processes can concurrently access their dispatch queues without interference (the intersection of all dispatch queues is null). To facilitate correct and efficient dispatching, the STT is sorted according to the scheduled start time of each task. This design provides a dispatcher with a constant time access to its dispatch queue to determine which task to execute next. Concurrent execution of the scheduler and the multiple dispatchers is achieved by reserving a set of tasks for each dispatcher. The scheduler is not free to reschedule the tasks reserved for the dispatchers. Thus each dispatcher has tasks to execute while the scheduler is attempting to reschedule in order to guarantee a new task.

The method of partitioning tasks between the scheduler and the dispatchers involves the calculation of a *cutoff line*. Once an upper bound of the scheduler's cost for guaranteeing a task is determined, this cost is added to the current time to determine the cutoff line. All tasks having a scheduled start time prior to the cutoff line are reserved for the dispatchers, and thus cannot be rescheduled.

The online guarantee is designed to allow concurrent operation with the multiple dispatchers. When a new, dynamic task arrives, the guarantee algorithm is invoked. The online guarantee does not alter the current schedule, it instead operates on copies of the task invocation information. This convention facilitates the return to the original STT if the guarantee fails.

3.1 Periodic Invocation of the Scheduler

Since the system processor is used for all system tasks, to ensure responsiveness for all system level activities, the scheduler as well as other tasks are invoked periodically. (In addition, if possible, the scheduler may also be invoked asynchronously upon the arrival of a new task.) Thus, the scheduler executes for, at most, a fixed amount of time, namely for the computation time allocated to it, every period. Periodic scheduler invocation affects the design of the real-time OS which bounds the task *guarantee* time, and the runtime (online) decision of how many additional tasks (extracted from the candidate queue, the queue of tasks waiting to be guaranteed) should be used for the next guarantee invocation.

Given that the scheduler has execution time which is O(N), (the number of tasks being processed), knowing the constant of proportionality and the fixed overheads, we can determine how many tasks can be guaranteed by the scheduler during each periodic invocation. Suppose this is Nmax. We call Nmax the "cap on the length of the STT". Suppose at a given time, the number of tasks already in the STT is S. Then at most Nmax - S tasks from the candidate queue can be considered for guarantee at this time. (Since the scheduler is invoked periodically, between two invocations, multiple task requests may get enqueued in the candidate queue.) Of course if not all Nmax - S tasks are guaranteeable, how to choose a subset of this set for subsequent attempts is an interesting question.

It is likely that the task invoking a nonperiodic task will impose a deadline not only on the invoked task, but also on the guarantee. In addition, some invokers may desire to know how long to wait to find out if the invoked task has been guaranteed or not. In the former case, whenever the scheduler is invoked, it has to examine the candidate queue to see if deadlines on the guarantee can be met given the discussion above. In the latter case, knowing the current length of the STT, etc., it is possible to determine the scheduler's response time.

3.2 Maximizing Concurrency between the Scheduler and Dispatchers

While the scheduler's execution time is a function of N, the number of tasks in the system (capped by Nmax), the dispatcher execution time need not be dependent on N. Because the worst case dispatching costs must be included in each task's worst case computation time, an efficient *worst case* design of the dispatcher is very important. In a multiprocessor implementation, worst case blocking time (in our case due to mutual exclusion with critical sections) can be the overwhelming cost of the dispatcher. Version 1 of the Spring multiprocessor OS uses dispatchers with constant worst case computation times i.e., the worst case computation times of tasks are *not* effected by the number of tasks in the system.

When an application task completes its execution, it must be deleted from the system. The most natural implementation is to have the local (running on the same processor where the task just completed) dispatcher delete the finished task from the system. This is not however the best implementation when the predictability of the multiprocessor OS is important, since, in order to maintain correctness with the scheduler, this design forces excessive mutual exclusive access to STT by the dispatchers. Specifically, if dispatchers were allowed to perform the deletions, the computation of the cutoff line would be required to be in a critical section (since pointers could become invalid during this computation). The computation of the cutoff line requires O(N) steps if the tasks are in a linked list, or O(log N) steps if the tasks were arranged contiguously. Thus, the scheduler could have locked the dispatch queue immediately prior to a dispatcher, causing the dispatcher to wait for an amount of time that is a function of the number of tasks in the system. This is unacceptable.

By having the scheduler, instead of the dispatcher, delete tasks from the STT, the worst case computation time of the dispatcher can be significantly reduced. The mechanics behind the convention of task deletions performed by the scheduler involve separate maintenance of dispatch queue pointers by the scheduler and the dispatchers. When a dispatcher notices that a task has finished, it implicitly marks the finished task by altering the head of the dispatch queue. The scheduler maintains a separate (shadow) copy of the dispatch queue head which is never altered by the dispatcher. When the scheduler is invoked, it first deletes all tasks which lie between the dispatch queue head and its shadow. Mutual exclusion is reduced to constant time – only modifications of the dispatch queue head need be done inside a critical section.

4. Real-Time Semaphores – Low Level Support for Predictability

The system components that are potentially the most elusive to guarantee predictability are those low level components which are shared by the multiple processors. On a multiprocessor, both shared memory and the shared bus connecting the processors fall into this category.

In a multiprocessor system, unless *bus access* time is bounded, any reference to remote memory cannot be predictable. For this and other reasons (discussed in [3]), any predictable real-time system which uses the asynchronous VME shared bus must be configured in round-robin mode. Round robin mode alone with processors busy-waiting on the semaphore (usually implemented with test-and-set) is however not sufficient to provide bounded waiting. It can be shown [3] that one or more processors can starve when two or more processors contend for a semaphore: It is possible for a subset of the processors to perpetually exchange the lock, starving one or more processors waiting for the lock.

To solve this problem, we have developed solutions for the construction of *real-time semaphores*[3] – semaphores which efficiently support bounded access. A software solution which improves the *bounded waiting* solution given in [1] as well as a hardware solution have been developed. The real-time semaphore is based on the P() and V() operations [2], using an extended test-and-set like operation, *test-and-set-or-branch*. The construction of real-time semaphores is

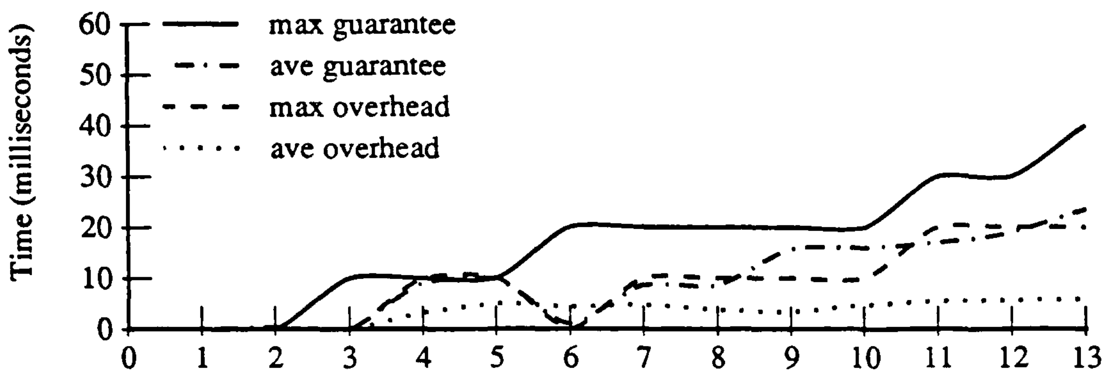


Figure 1: Spring Version 1 Performance.

based on the *Deferred Bus Theorem* (see [3] for the proof of the theorem):

> If the total worst case non-bus master time of the busy-wait loop (in P()) is less than the best case bus master time of the release instruction, and if processor p_j is the closest processor (in the round robin ordering) busy-waiting for semaphore s when processor p_i releases s (in V()), then p_j will be the next processor to acquire s.

Operations for enforcing mutual exclusion operations such as P() and V(), if constructed in a bounded fashion, can provide the framework for other, higher level, bounded operating systems primitives. This boundedness forms a basis for the predictability of the Spring real-time multiprocessor OS.

5. Performance

The performance of the scheduler (running in a 16 MHz. 68020) in Spring version 1 is illustrated in figure 1. Both the average and worst case computation times of the guarantee algorithm and the overheads are plotted. The costs of the guarantee algorithm are separated from the costs of the overheads, the total scheduler cost being the sum of the two. The overheads consist of scheduler activities before and after invocation of the guarantee algorithm (such as the computation of the cutoff line and task deletions). The guarantee algorithm, as described in [4], is invoked with no backtracking for a system with seven resources.

As discussed in section 3.1, the periodic invocation of the scheduler imposes a fixed computation time for the scheduler to run. Depending on the selected period and length of this fixed computation time, a cap on the maximum number of tasks which are guaranteed in this fashion (using the heuristic guarantee algorithm described in [4]) will be derived. Practical optimizations are currently underway, and include alternative scheduling algorithms, and restricted data structure access. One scheduling optimization would be, instead of performing a total reschedule of all tasks in the system, attempting to insert a single task into the existing schedule. The examination of only portions of sorted system tables is an area of optimization pertaining to restricted data structure access. By speeding the guarantee, these optimizations may allow us to deal with more tasks than the more general techniques which have been implemented.

6. Conclusion

Our approach to constructing a real-time OS is to achieve predictability from the bottom up. We have discussed how bounded access to a shared bus facilitates the construction of real-time semaphores. Real-time semaphores in turn form a foundation for the construction of a *concurrent*, predictable real-time multiprocessor OS. At the next level, the predictability of user level tasks is facilitated by the predictable OS. Subtleties arising in supporting the online guarantee complicate the construction of a predictable multiprocessor OS which is concurrent. These subtleties are resolved in part by offloading activities from dispatcher to the schedulers, integrated with judicious use of critical sections by the real-time OS. The feasibility of this approach has been demonstrated with the shared bus multiprocessor implementation of Spring version 1.

References

[1] J. E. Burns. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[2] E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5), May 1968.

[3] L. D. Molesky, C. Shen, and G. Zlokapa. Predictable Synchronization Mechanisms for Multiprocessor Real-time Systems. Technical Report 89–106, University of Mass., November 1989.

[4] K. Ramamritham, J. A. Stankovic, and P. Shiah. O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems. In *the 9th International Conference on Parallel Processing*, June 1989.

[5] J. A. Stankovic. Misconceptions About Real-Time Computing. *IEEE Computer*, 21(10), Oct. 1988.

[6] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Operating Systems. *Operating Systems Review*, 23(3), July 1989.

LynxOS: UNIX Rewritten For Real-Time

Inder M. Singh, Mitch Bunnell
Lynx Real-Time Systems, Inc.

Introduction

During recent years, there has been a major movement in the computing world towards standard, open systems based on the UNIX operating system. Key attractions of standard, open systems include vendor-independence, protection from technological obsolescence, preservation of software investment, availability of off-the-shelf applications software from independent suppliers, availability of trained programmers familiar with standard environments, and connectivity between various dissimilar computers within the user organization.

By contrast with this trend towards open, standard platforms, the world of real-time computer applications is fragmented: several proprietary operating systems and kernels are being used with no single software environment accounting for a significant fraction of the applications.

Real-time applications typically require computers to interact with external equipment in "real-time"; this usually involves being able to respond very quickly to external events with the requirement for response time varying from a fraction of a second down to microseconds. Typical real-time applications involve control or monitoring of systems by computers, the collection of data in real-time, or real-time simulation of complex systems.

In more traditional computer applications that access data files, print reports or deal with interactive users, occasional delays of several seconds or longer are usually acceptable. With real-time applications, on the other hand, the computer needs to respond within a certain amount of time, often in the millisecond or microsecond range, in order for the system to function correctly. Most important of all, this response delay must be deterministic.

There has been a strong and growing interest in UNIX for these real-time applications. All of the benefits of open systems mentioned above are equally applicable to real-time applications. UNIX, which is the only "open" multi-tasking operating system that has been ported to a large number of CPU architectures and is available from a large number of vendors, is a very attractive candidate. The development environment of UNIX is also quite popular among software developers.

Problems With UNIX For Real-Time

Unfortunately, UNIX was never designed for real-time and while it is a fine general-purpose time-sharing system, it has serious limitations for real-time applications. It introduces large delays for high priority tasks that need to respond to events external to the computer, and moreover, this delay is highly variable and unpredictable. Since UNIX is designed to support a number of different users at the same time, it tries to be fair to all of them, and it does not allow direct control of critical shared resources. In a real-time application, by contrast, the programmer or user needs to have much more direct control over the system, including absolute control over priorities and which task can run on the CPU at a given time. Most important of all, the system must be able to respond to critical external events very quickly and in a predictable amount of time.

Earlier Attempts At Real-Time UNIX

In spite of the above limitations of UNIX for real-time, the attractiveness of UNIX is so great that many attempts have been made to address this "real-time UNIX" need. However, until now, none of them have been entirely satisfactory.

Several efforts have been directed at modifying the UNIX system from AT&T to improve its suitability for real-time applications. In real-time UNIX systems of this kind, the scheduler is typically reworked to implement a scheduling algorithm suitable for the needs of real-time tasks, and preemption points are inserted into the longer system calls to reduce the latency for context switches. Several other features of importance for real-time applications are often added such as memory locking, timer support and others. Finally, the system is put through a process of performance tuning to improve system responsiveness for real-time.

Requirements For Real-Time UNIX

For a real-time UNIX compatible system to gain widespread acceptance, and to provide all the potential advantages of an open, standard system, it must satisfy a basic set of requirements in two major areas: UNIX compatibility and real-time performance:

UNIX Compatibility

Such a system should behave, look and feel like UNIX; moreover, it should be positioned similarly to UNIX in the sense of being a truly open, portable system that will follow the future evolution of UNIX including supporting mainstream UNIX extensions in areas like the user interface and networking.

Full UNIX compatibility means the ability to run any program that is written to be portable across UNIX systems; such a system should provide at a minimum source level compatibility with the standard AT&T System V Interface definition (SVID) and the emerging IEEE POSIX 1003.1 standard. Source compatibility with the Berkeley 4.3 flavor of UNIX is also very desirable. Future versions will have to track the dominant UNIX standard as represented by system V.4 and beyond from AT&T and future offerings from the OSF as well as the POSIX 1003.4 specification for real-time extensions to UNIX when that standard is adopted. In addition to the kernel, the shell and all important utilities have to be available and be compatible, although the requirements are not quite as rigid in terms of applications portability.

Source compatibility essentially means supporting all UNIX system calls either directly or through library functions. Beyond source compatibility, to provide the full benefit of a standard, open system, it is important to support binary (or executable) compatibility across implementations on the same CPU architecture where an applications binary interface (ABI) has been defined such as for the 80386 and the 680x0. Binary compatibility provides the opportunity for "shrink wrapped software" from third party software manufacturers, which can be run by the user without having to obtain the source and recompile it.

Real-Time Performance (And Functionality)

To gain widespread acceptance, real-time UNIX solutions have to provide the real-time performance that users expect from the leading proprietary solutions. So long as this performance gap is large, the perception that UNIX is not adequate for real-time will persist and users will continue to use proprietary solutions.

Real-time applications require a computer to react to periodic or external events within a bounded amount of time. The real-time response of the program running under an operating system depends not only on the speed of the hardware and the efficiency of the compiler, but also on the interrupt response and task response provided by the operating system. Interrupt response is the delay between an external event and the invocation of the associated interrupt routine. The interrupt routine records the event, buffers data, and requests that the task that must respond to the event be scheduled to run. If this is the highest priority task runnable, it is dispatched after a context switch. The task response is the delay between the event and the task actually beginning its execution. The task response is important because it is the task that can save data to mass storage, perform control strategies, update operator displays, etc. The interrupt response and task response delays need not only to be small but also deterministic -- it is the worst case value of these parameters that have to be used by the system implementor.

Real-time applications often have to interact with many external devices such as sensors, analog/digital convertors, motors, solenoids, etc. Thus a real-time system should provide an efficient interface to these devices. It should be easy to write device drivers in a higher level language such as C. Overheads for accessing these devices should be kept to a minimum, and interrupt latency should be low.

The tasks implementing a real-time application must be able to communicate efficiently with each other to coordinate their operation. The system should support shared memory segments across tasks, as well as a rich set of interprocess communication mechanisms such as semaphores, signals, named pipes and sockets. Since many applications require an embedded system without a hard disk, it must be possible to put both the operating system and the application in ROM.

LynxOS: Real-Time UNIX From The Ground Up

With the above requirements in mind, the design of LynxOS focussed on the twin objectiveness of preserving full UNIX compatibility while providing superior real-time response and enhanced I/O facilities.

Designing Real-Time Performance Into UNIX

The design of LynxOS is heavily focussed on achieving excellent real-time response without compromising on UNIX compatibility. LynxOS uses a method of process scheduling more applicable to real-time, and allows preemption in the kernel to improve task response. Task and interrupt response are further improved by performing any extended asynchronous processing at normal process priority levels instead of interrupt levels.

Like UNIX, LynxOS is a multitasking system with priorities associated with tasks; the task with the highest priority is normally the one that runs on the processor. Like most time-sharing systems, the UNIX OS determines this priority level for each task. The user may provide a priority level (nice), but this is only used as a guideline -- the kernel uses this information along with other information such as the CPU time used by the task, what I/O the task is using, etc., to determine the actual priority to be assigned to the task. In a real-time system like LynxOS, on the other hand, a critical task must be dispatched as soon as possible after the notification of an event. The user must be able to set the absolute priority of the task and not have it modified by the OS.

Kernel Pre-emption

When a high priority task is ready to run, a lower priority task running on the processor has to be preempted. However, the operating system has to protect its data structures from being partly accessed by one task, then corrupted by a second task that has preempted the first. In the design of UNIX, this problem is solved by disabling preemption whenever the system is executing in the kernel. A low priority task executing a system call cannot be preempted by a higher priority task until it completes the system call or is blocked for I/O. Many system calls, such as FORK, can take a long and unpredictable amount of time. This is one of the major sources of the poor and unpredictable task response time of UNIX.

The LynxOS kernel is designed to be preemptible at almost any time to avoid this problem; two techniques are used to protect kernel data structures. Most kernel data structures are specifically designed to be accessed during very brief periods, and they are left in a safe state at all other times; it is only during these brief access periods that task switches are disabled. Where this is not possible, only parts of data structures are locked, using counting semaphores, so that no other tasks can corrupt them.

Process Synchronization

In UNIX, when a signal has to be delivered to a process, or the process needs to be pre-empted, this is accomplished by invoking an asynchronous software trap (AST), which is a software settable interrupt at a priority level below all hardware interrupts. The AST is taken when the process enters user mode, after all hardware interrupts have been handled.

In LynxOS, two ASTs are used to invoke delivery of signals and task pre-emption, respectively. The signal AST (AST1) is at the lowest priority, and is disabled during system calls so that it is only taken in user mode. Since the LynxOS kernel is pre-emptible, the pre-emption AST (AST2) is at the next higher priority level, and it is normally enabled during a system call. To protect a critical kernel data structure, the processor priority is simply raised above the level of the AST2 while the data structure is being accessed. Where the underlying hardware does not support ASTs, they are simulated in software.

When a kernel data structure that was locked is released, the UNIX kernel wakes up every task that was waiting on the associated semaphore. By contrast, the LynxOS kernel only wakes up the process at the highest priority waiting on the semaphore; if there is more than one waiting task at this priority level, the first one that waited on the semaphore is selected.

Impact Of Interrupt Handlers On Real-Time Response

The handling of interrupts is another area that has major impact on the responsiveness of the system. Since all interrupt handlers run at a higher priority than regular tasks, the worst case task response of the system can be no better than the execution time of the longest interrupt handler. In time sharing systems like UNIX where all asynchronous processing is done in interrupt handlers, this can be quite long. The standard UNIX terminal driver does all its line editing functions at interrupt level. On a system with a memory mapped screen, such as the console on an 80386 AT-compatible PC, the entire screen is scrolled when a newline is echoed in an interrupt routine. A streams driver can improve this somewhat. A stream is an asynchronous computation path that runs at the level of the lowest priority interrupt on the system. This improves the response for higher priority interrupts, but this processing still has a higher priority than any task.

LynxOS uses a technique based on kernel threads to minimize the impact on real-time responsiveness of interrupt handlers that require significant amounts of processing. The interrupt handler itself only performs tasks that are essential, such as buffering critical data or handshaking with I/O devices. All other processing, which would be performed by the interrupt handler in a normal UNIX system, is executed by a kernel thread, or lightweight task. Kernel threads have priorities like user tasks and they are scheduled along with normal tasks. Thus asynchronous processing can be done at a lower priority than critical real-time tasks. A kernel thread is used, for example, for asynchronous terminal processing; its priority is set to the same level as the highest priority task using the terminal. With the LynxOS approach, high priority tasks implementing real-time functions run unaffected by line editing at terminals being used for lower priority applications such as program development.

Expanding The UNIX I/O System

Time sharing systems including UNIX are designed assuming a small number of devices that are relatively stable: a typical system supports a few disk drives, some terminals and a printer. By contrast, real-time systems must support a much wider variety of devices including digital I/O boards, analog/digital converters, servo motor controllers, etc.. Further, users frequently need to interface to new devices.

LynxOS extends the UNIX unified I/O system to provide a common interface for many additional classes of devices. For example, LynxOS includes a utility called SAIO, similar to UNIX's STTY, that provides a uniform way to control analog/digital convertor boards.

In many real-time systems there are a large number of input channels for a given class of device. Having an open file descriptor and reading from each channel is inefficient and runs into limits on the number of open channels for a task. In a system with 40 temperature sensors, it would be slow and messy to have 40 open files. Plus, in many control applications, inputs must be read simultaneously. LynxOS solves these problems by allowing channels to be grouped. All channels in a group can be read and written simultaneously through a single file descriptor.

LynxOS device drivers can be dynamically loaded and unloaded, greatly facilitating the development of drivers for new devices.

Contiguous Files

To provide the high disk throughput required by many real-time applications such as high speed data acquisition, LynxOS supports contiguous files. The space for a contiguous file is preallocated within the normal file system at file creation time. All subsequent accesses to the file use normal UNIX calls like read and write; however these calls perform raw I/O to the file bypassing the cache and the data is transferred directly using DMA from the disk interface into the user memory. This makes it easy for users to actually achieve the maximum disk throughput provided by the hardware for real-time applications.

Non-volatile Disk Cache

LynxOS uses a disk cache just like UNIX to provide fast throughput for multiple tasks accessing disk files. Since the cache uses a write to, rather than a write through, algorithm for maximum performance, UNIX must flush the cache every 30 seconds or so just in case there is a power failure. Even so, data can be lost and the disk corrupted. So when a UNIX system is brought up it does an exhaustive check of the file system on disk and corrects any problems. This can be a serious problem in real-time applications where the system must come back on line as quickly as possible. LynxOS supports a battery backed-up disk cache, if supported by the underlying hardware, to prevent a loss of information. In-core inodes are saved in this cache in addition to recently used disk cache blocks. When LynxOS is booted, it checks the cache and knows the exact state of the cache prior to the last time that the system was turned off. Only if the system was in the middle of a disk write does it do a file system check. So in most cases, the system is up and running right away without the 5-15 minute delay of a typical UNIX system. This mechanism also eliminates the need for periodic disk synchronization.

ROM-based Systems For Embedded Applications

For many real-time applications, a hard disk or network is unavailable or unsuitable, requiring the system to operate out of ROM. A ROM-based system provides greater ruggedness than disk-based systems, as well as savings in space, power and cost. A ROM-based system also minimizes boot-up time. Standard UNIX requires a block device such as a hard disk for its file system and for booting; also the kernel is rather large for most ROM-based applications, typically requiring a megabyte or more of memory.

LynxOS is designed to operate out of ROM, if required, and is compact enough to be placed in ROM along with applications code. Depending on the configuration, the LynxOS kernel requires from 130 to 160 kilobytes of memory for code and initialized data. One can build embedded systems with as little as 256 Kb of ROM and 256 Kb of RAM. A ROM-disk driver allows a ROM to contain the LynxOS file system and to be used as the default root and boot device. In addition, both the kernel and executable programs can be executed directly from ROM. A special magic number identifies such programs; when the file is executed, the data segment is loaded into RAM and space for uninitialized data is allocated, but the code segment in the ROM is mapped directly into the process's address space.

Conclusion

The LynxOS project has demonstrated the feasibility of using standard, open systems for real-time applications, and it also demonstrates a major potential advantage of the standardization activity that is under way for UNIX through the IEEE 1003 committee and other bodies. A real standard supports multiple implementations from different vendors to the same interface specification. This gives the users multiple choices and promotes competition between vendors which leads to the best value for the user. It also provides alternative versions optimized for different environments.

Research in Time- and Error-Constrained Database Query Processing[†]

Gultekin Ozsoyoglu[+], *Z. Meral Ozsoyoglu*[+], *and Wen-Chi Hou*[-]

ABSTRACT

One may define a *time-constrained database* as a database that has strict timing constraints in (a) responding to queries, (b) processing transactions, and in (c) database maintenance such as integrity enforcement, view management, and database insertions, deletions, and updates. A *time-constrained query* has the form of "get the information x in no more than t time units". Presently, there is no comprehensive time-constrained DBMS methodology available.

Another type of queries is the class of error-constrained queries. A typical query may be "evaluate the query Q with the error function e(Q) being less than a certain bound". In such a case, the user implicitly permits a reduction in the evaluation time of the query by agreeing to a bounded error in the answer of the query. Time- and error-constrained queries may occur in centralized/distributed, and single-user/multiuser environments.

This position paper describes our research directions towards a general methodology for processing arbitrary time- and error-constrained database queries.

1. Introduction

As the database technology matures, it is being integrated into larger systems where there are new requirements on the database management component of the system. We think that real-time, near-real-time, and time-constrained computing systems can benefit from having a database management system (DBMS) as a component, and have new DBMS requirements. Presently, there is no comprehensive real-time, near-real-time, or time-constrained DBMS methodology available.

One may define a *time-constrained database* as a database with the new requirements that it has strict (possibly, real-time) timing constraints in (a) responding to queries, (b) processing transactions, and in (c) database maintenance such as integrity enforcement, view management, and database insertions, deletions, and updates. A *time-constrained query* has the form of "get the information x in no more than t time units". Below we list two application areas that may utilize time-constrained databases.

Area 1. *Databases Used in Manufacturing Environments.* The so-called "automated factory of the future" uses various databases at different levels of factory automation. At the lowest level, the factory floor level, there are *programmable logic controllers* (PLC) which are special-purpose computers (or, sometimes, modified personal computers) that control manufacturing processes using the present and the past history of some "input" manufacturing processes. Presently, PLCs maintain and manipulate data about input processes in very rudimentary forms because the data manipulation is explicitly coded by the users, and there are strict timing constraints on responses. Clearly, a main-memory time-constrained database system with the ability to respond to time-constrained queries can increase the capabilities of PLCs, and ease users' jobs.

Area 2. *Databases Used in Scientific Applications.* In some scientific experiments and applications, the data

† This research is supported by the National Science Foundation under Grants DCR 8605554, and IRI-8811057.
+ Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106
- Department of Computer Science, University of Southern Illinois at Carbondale, IL 62901

gathered is so large that "processing the data on-the-fly" during the execution of an experiment/transaction is needed. Processing the data on-the-fly puts a time constraint on query evaluation, i.e., queries must be processed in fixed time-units and, for some experiments, in real-time. Also note that, in such an environment, the queries are repetitive, and occur periodically.

Time-constrained queries that we have described above have the "hard" time constraint which is not to be missed. There are other applications in which the time constraint is "soft", i.e., the time deadline can be missed in evaluating the query, but as time passes, the value of the response to the query diminishes. Another type of queries is the class of error-constrained queries. A typical query may be "evaluate the query Q with the error function e(Q) being less than a certain bound". In such a case, the user implicitly permits a reduction in the evaluation time of the query by agreeing to a bounded error in the answer of the query. Time- and error-constrained queries may occur in centralized/distributed, and single-user/multiuser environments.

For the last two years, we have been working on the problem of processing time-constrained *aggregate* relational queries. This paper summarizes our approach for building a methodology to process

(a) time constrained ad hoc queries that produce relations,

(b) time constrained periodically occurring queries, and

(c) error constrained queries.

2. Previous Work on Aggregate Queries

Consider a single-user database, and the problem of processing an aggregate database query within a given time quota. That is, we would like to process the query "evaluate $f(E)$ within T time units" where f is an aggregate function (e.g., SUM, COUNT, AVERAGE) and E is an arbitrary relational algebra expression (of the relational database model).

Our approach is as follows. First we perform an a priori analysis and locate "good" (i.e., unbiased, consistent, etc.) estimators $\hat{f}(E)$ of $f(E)$ for various E expressions. When the query is posed, due to the lack of time, instead of evaluating $f(E)$, we choose an estimator, say $\hat{f}(E)$, and evaluate the statistical estimator $\hat{f}(E)$ by sampling from operand relations in E. For the evaluation, given the time quota T, we decide about the sizes of the samples, obtain the samples, and evaluate the estimator $\hat{f}(E)$. If, at the end of the evaluation of $\hat{f}(E)$, there is time left then we perform a second stage of obtaining additional samples and improving the estimate for $f(E)$. We continue improving the estimate by additional stages until the time quota is completely used.

The statistical approximation approach summarized above needs to be revised when the database system is a multi-user and/or a distributed system. In such cases, it may not be possible to satisfy the time constraints of all user queries, and a compromise is needed to decide which user queries get processed within their time quotas. Also, when the query is not an aggregate query, different approaches are needed. These new approaches may or may not use estimation.

In [HoOT 88, HoOz 90], we develop a general methodology to obtain consistent and unbiased estimators for the aggregate query $COUNT(E)$, where E is an arbitrary relational algebra expression. The basic theoretical framework developed is based on the simple random sampling method, and then extended to a cluster sampling plan for efficiency considerations. Extensive performance evaluation of the estimators using a prototype system are reported in [HoOz 90].

Given a $COUNT(E)$ query and a time quota, we discuss in [HoOT 89] the issues of

(i) when to stop processing the query (i.e., the *stopping criteria*) to meet the time constraint, and

(ii) how to use the time quota efficiently (i.e., the *time control strategies*).

The time-control algorithm determines the sample size from each operand relation such that the estimator $\hat{f}(E)$ can be evaluated with a desired probability within the given amount of time quota. [HoOT 89] introduces statistical and heuristic time control strategies, and compares them for the control of the risk of overspending the time quota.

A disk-based database management system that was previously developed in our department is revised, and used [Hou 89] in implementing and evaluating the methodology of [HoOT 88, HoOT 89, HoOz 90]. We have just revised and converted the disk-based system into a main-memory-only database management system [Liu 89], and started testing our methodology in this environment.

3. Related Work

In the last two years, a number of research results about "real-time databases" have started to appear in the literature. In general, these publications define a "real-time database" as a database in which transactions have severe response time constraints, that is, they must be completed within their deadlines. As the transaction completion by the deadline is dependent on the workload of the system and is not possible for all transactions, the goal changes. For example, in [AbGM 88], the goal is to minimize the number of transactions that miss their deadlines. Below, we briefly summarize the approaches taken by different researchers.

(a) Redesigning the conventional database systems or their components to obtain high performance [Sing 88],

(b) Introducing parallel query processing, filtering and data flow control to improve the performance [Bult 88],

(c) Revising techniques from operating systems in the areas of CPU scheduling, transaction management, deadlock management, buffer management, and disk I/O scheduling.

In *CPU scheduling*, the algorithms investigated [HSTR 89, StZa 88, AbGM 88, AbGM 89] include most-critical-transaction-first, by-criticalness-and-deadline, least-slack, and first-come-first-serve. In the case of resource conflicts among transactions, the *conflict resolution techniques* investigated [AbGM 88, AbGM 89, BMMU 89, McDA 89, Daya 88, Sing 88, LinL 88, HSTR 89, StZa 88] are first-come-first-serve, wait, wait-promote, high-priority, conditional-restart, dynamic parameters, virtual clock, deadline-first-then-criticalness, deadline-criticalness-remaining-execution-time, criticalness-only, value-function, and nonserializable-but-consistent. As *conflict avoidance techniques*, [BMMU 89] investigates preanalyzed transaction classes. As *transaction wakeup policies*, [HSTR 89] investigates max-virtual-clock-time, max-combined-parameters, min-deadline, and highest-criticalness policies. For *transaction commit policies*, [AbGM 88] considers the policy of log-on-separate-disk-dirty-pages-in-buffer. As *transaction restart policies*, [HSTR 89] investigates nonzero-value-restart, restart-with-increased-priority, and restart-if-feasible. For *deadlock resolution techniques* among transactions, [HSTR 89] investigates transaction abortion for transactions with the longest-deadline, earliest-deadline, least-criticalness, or tardy/feasible-with-least-criticalness. Among *buffer management techniques*, [CaJL 89] considers priority-LRU, and priority-DBMIN, and [AbGM 88] considers random-replacement. For *disk scheduling techniques*, [AbGM 88] investigates highest-priority, and [CaJL 89] investigates scan-within-priority-groups.

4. Current Work

With the exception of section 4.5., we consider only centralized, single-user databases. We think that this is a necessary step which should precede the investigation of real-time, centralized/distributed, multi-user da'abases.

4.1. Queries that Request Relations

If the response to a query is not a single value (as in aggregate queries) but a set of tuples, what type and how much information can we give to the user? Three possible approaches are

(a) to return "profile" statistics about expected values for output attributes,

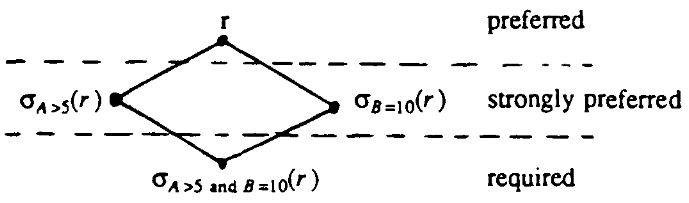
(b) to estimate either univariate or multivariate distributions of output attributes in a functional form with estimated parameters ,

(c) develop incremental, "subquery" evaluation methodologies.

Below we elaborate only on (c), which is the approach that we are investigating.

A. Relation Fragmentation Lattices

Consider a relation r with attributes A, B and other attributes. We fragment r using a set of selection conditions, i.e., we form a *relation fragmentation lattice* such that each fragment is assigned a priority. A very simple lattice is given below.



where σ denotes the selection operation of the relational algebra, and "required", "strongly-preferred", and "preferred" denote the priorities. Relation fragmentation lattices are maintained independent of users' queries.

Our approach is that, in a given query with relation r, we use fragments of r (or their combinations) to evaluate the query in an incremental fashion. To illustrate, assume that the relation r is split into two fragments, say fr_1 and fr_2, and the query Q(r) is converted into Q'(fr_1) ∪ Q"(fr_2). We make sure that, by our choices of Q' and Q", the output tuples of Q' and Q" are disjoint, and we avoid duplicate tuple elimination in the set union operation ∪. (Duplicate elimination in the set union operation of relational algebra is very costly (since it involves the sort operation).) This evaluation process is guided by

1) an optimization process involving levels in hierarchies, e.g., required/strongly-preferred/preferred fragments,

2) multiple relation fragmentation lattices, and

3) "answer categories" with respect to certain "error functions", e.g., better/locally-better/globally-better answers.

B. Incremental Evaluation of Periodically Occurring Queries

We devise new techniques for "periodically occurring queries", i.e., those queries that are repetitively executed at certain time points. In such cases, changes in each input relation r of a query Q are stored in Δr, an *incremental relation*. Then, the query Q(r ∪ Δr) is converted into the union of two (disjoint) queries Q(r) and Q'(r, Δr), where Q(r) is already computed in the previous step, and Q'(r, Δr) can be computed fast and within the time constraint.

When Δr contains only tuple insertions and the query Q is monotone (i.e., for larger relations, Q produces larger output), incremental evaluation of Q is straightforward. When Q is nonmonotone (e.g., when Q contains the set difference operator), new techniques need to be developed.

C. Subexpression Fragmentation Lattices

In addition to relation fragmentation lattices, the user of a query may choose to define a fragmentation lattice involving the attributes of a subexpression of the query (e.g., after a join operator in the query). This permits an incremental time control in processing the rest of the query.

Unlike relation fragmentation lattices, subexpression fragmentation lattices are built during the query evaluation time since we assume that the queries are generally ad hoc.

D. Incremental Maintenance of Lattices

A methodology is being developed such that the relation fragmentation lattices are maintained (independently of queries) using incremental evaluation techniques. Also, when a query is periodically occurring, its subexpression fragmentation lattices should also be maintained by incremental evaluation techniques. Please note that the techniques to incrementally maintain relation and subexpression fragmentation lattices are different since the latter involves arbitrary relational algebra operations and the former involves only the selection operator.

E. Output Tuple Classifications

When a query cannot be evaluated completely at the end of a given time constraint, we can still classify and use some of the "currently produced" tuples. One classification may contain the "definitely in the output" and the "possibly in the output" classes. For example, consider the query Q which is of the form $E_1 \cup (E_2 - E_3)$, where E_1, E_2, and E_3 are relational algebra expressions. Assume that, at the end of the time quota, E_1 and E_2 are completely evaluated, but $E_2 - E_3$ is partially evaluated (and is therefore useless). In such a case, the tuples of E_1 are of type 'definitely in the output", and the tuples of E_2 are of type "possibly in the output". Another classification may be a probabilistic classification which attaches "output inclusion" probabilities to tuples.

4.2. Estimating Error-Constrained Queries

Consider a situation where a query response has to satisfy a given error constraint. The issues to be investigated are how to determine the sample sizes and how to satisfy the error constraint requirements efficiently. The estimation of the sample size needs the knowledge of the characteristic of sample units. Unfortunately, this information is usually unknown.

Similar approaches to [HoOT 89], e.g., using prestored statistics and using run-time estimations need to be considered for estimating the sample size. For the run-time approach, instead of assuming a large variance of sample units, we may assume a minimum ("reasonable small" perhaps) variance at the first-stage evaluation.

Taking an equal size of sample from each input relation may not be the best way to satisfy error constraints. We need to identify those factors that may influence the precision of estimations. For example, a possible candidate may be the selectivity of an operation.

4.3. Estimation of COUNT(E) Queries

A. Different Sampling Methods

Systematic sampling [Coch 77] selects sample units at a fixed interval. It has the advantages of being simple and producing more evenly spread samples than the simple random sampling (and hence, probably has a better precision). The disadvantages are (1) there is no trustworthy method for estimating the variance of the estimated mean, (2) the performance of systematic sampling depends on the properties of the population. It may be extremely precise for some populations and it may be less precise than simple random sampling for others.

The second- (or multi-) stage cluster sampling is an alternative when a disk block contains tuples with similar or duplicate information. That is, instead of using all the tuples in a block, only a random sample of the tuples in a block may be used for economic reasons.

In stratified random sampling, the population is first divided into strata based on certain properties, i.e., the values of certain set of attributes that are of concern to us, and then a random sample is taken from each stratum. The stratification may produce a gain in precision in the estimates of characteristics of the whole population.

B. New Estimators

The ratio estimator [Coch 77] usually gives biased estimates. However, it is quite promising since it gives a smaller variance of the estimate (when the size of sample units are different) compared to the unbiased one in [HoOT 88]. Different sizes for sample units may occur due to insertions, deletions or variable sizes of tuples in secondary storage blocks.

In [HoOz 90], estimation of the number of classes of identical tuples in a multi-set" is needed after a projection operation of relational algebra. To this end, we have used Goodman's estimator [Good 49]. When the sample fraction is low, Goodman's estimator is unstable for a population with heavy duplicates. We need to find stable and good estimators for the projection operation.

4.4. Estimators for Other Aggregate Functions such as Sum or Average

The estimation of SUM(E) can be done exactly the same way with the estimation of COUNT(E)-except when there is a projection operation of the relational algebra. Projection may produce (and later eliminate) duplicates, and duplicate tuples make the inclusion probabilities of different set of tuples in the sample unequal, and create problems in counting the total number of resulting tuples.

As for AVERAGE, estimators for union, difference and projection operations of the relational algebra are not straightforward since duplicate tuples make the inclusion probabilities of different tuples unequal. Also, the estimation of an arbitrary relation algebra query containing union and difference operations needs further attention since the principle of inclusion and exclusion (heavily used in [HoOz 90]) is not valid for AVERAGE.

Nonparametric procedures, e.g., sign test and Wilcoxon sign-rank test, are usually used for estimating MEDIAN. The sign test can be used for any distribution while the Wilcoxon sign-rank test is valid for symmetric probability distribution. We need to investigate how these techniques can be incorporated into the estimation of MEDIAN of attribute values of a relational algebra query.

4.5. Centralized, Multi-User Databases

For real-time, multi-user databases, we think that the most promising approaches are [AbGM 88, AbGM 89], which uses heuristics to minimize the number of transactions that miss their deadlines, and [HSTR 89] which uses protocols to handle CPU scheduling, data conflict resolution, etc. These techniques need to be merged with the "query level" controls introduced above, and their performances need to be compared.

5. References

[AbGM 88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation", Proc., *VLDB Conference*, Los Angeles, CA, 1988.

[AbGM 89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data", Proc., *VLDB Conference*, Amsterdam, the Netherlands, 1989.

[BMMU 89] Buchmann, A.P., McCarthy, D.R., Hsu, M., and Dayal, U., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control", Proc., *IEEE Data Engineering Conf.*, Los Angeles, Ca, 1989.

[Bult 88] Bultzingsloewen, G. et al, "KARDAMOM--A Dataflow Database Machine for Real-Time Applications", *ACM SIGMOD RECORD*, Guest Ed. Sang H. Son, March 1988.

[CaJL 89] Carey, M.J., Jauhari, R., and Livny, M., "Priority in DBMS Resource Scheduling", Proc., *VLDB Conference*, Amsterdam, the Netherlands, 1989.

[Coch 77] Cochran, W.G., *Sampling Techniques*, Third Ed. John Wiley & Sons, 1977.

[Daya 88] Dayal, U. et al, "The HiPAC Project: Combining Active Databases and Timing Constraints", *ACM SIGMOD RECORD*, Guest Ed.: Sang H. Son, March 1988.

[Good 49] Goodman, L. A., "On the Estimation of the Number of Classes in a Population", Ann. Math. Stat., Vol. 20, 1949.

[HoOT 88] Hou, W-C., Ozsoyoglu, G., and Taneja, B. K., "Statistical Estimators for Relational Algebra Expressions", Proc., *ACM Symposium on Principles of Database Systems*, Austin, TX, 1988.

[HoOT 89] Hou, W-C., Ozsoyoglu, and Taneja, B.K., "Processing Aggregate Relational Queries with Hard Time Constraints", Proc., *ACM SIGMOD Conference*, Portland, OR, 1989.

[HoOz 90] Hou, W-C., and Ozsoyoglu, G., "Statistical Estimators for Aggregate Relational Algebra Queries". To appear in *ACM Transactions on Database Systems*, 1990.

[Hou 89] Hou, W-C., "Relational Aggregate Query Processing Techniques for Real-Time Databases", PhD Thesis, CWRU, May 1989.

[HSTR 89] Huang, J., Stankovic, J.A., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing", Unpublished Manuscript, Univ. of Massachusetts at Amherst, 1989

[LinL 88] Lin, K-J., and Lin, M-J., "Enhancing Availability in Distributed Real-Time Databases", *ACM SIGMOD RECORD*, Guest Ed.: Sang H. Son, March 1988.

[Liu 89] Liu, Y-M., "A Main-Memory Real-Time Database Management System--Implementation and Experiments", MS Thesis, CWRU, July 1989.

[McDA 89] McCarthy, D.R., and Dayal, U., "The Architecture of an Active Data Base Management System", Proc., *ACM SIGMOD Conference*, Portland, OR, 1989.

[Sing 88] Singhal, M., "Issues and Approaches to Design of Real-Time Database Systems", *ACM SIGMOD RECORD*, Guest Ed.: Sang H. Son, March 1988.

[StZa 88] Stankovic, J., and Zhao, W., "On Real-Time Transactions", *ACM SIGMOD RECORD*, March 1988.

Scheduling Real-time Transactions in Distributed Database Systems†

Sang H. Son
Juhnyoung Lee

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

1. Introduction

Real-time databases are used in a wide range of applications such as aircraft tracking and the monitoring and control of modern manufacturing facilities. In a real-time database context, concurrency control protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. One of the most important requirements of real-time database systems is to complete as many transactions as possible without violating their timing constraints [Son88].

Concurrency control protocols control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database. Serializability is a widely accepted notion of the definition of correctness for concurrency control in database systems. A *scheduler* in database systems is entrusted with the task of enforcing the serializability constraints. It accepts database operations from transactions and schedules them appropriately for the data manager. To satisfy both data consistency and real-time constraints in real-time database systems, there is the need to integrate concurrency control protocols with real-time scheduling protocols.

Real-time scheduling protocols address the problem of meeting the specified timing constraints. Satisfying the timing constraints of real-time systems demands the scheduling of system resources according to some well-understood algorithms so that the timing behavior of the system is understandable, predictable, and maintainable. The goal of most scheduling problems is to find optimal static schedules which minimize the response time for a given task set. In many real-time systems, however, there is generally no incentive to minimize the response time other than meeting the deadline. Real-time systems are often highly dynamic requiring on-line, adaptive scheduling algorithms. Such algorithms must be based on heuristics since these scheduling algorithms are NP-hard [Stan88]. In these cases, the goal is to schedule as many jobs as possible, subject to meeting the task timing constraints. Alternative schedules and/or error handlers are required and must be integrated with the on-line scheduler.

While there is progress in both concurrency control of transactions and real-time scheduling, very little is known about the integration between concurrency control protocols and real-time scheduling protocols. Real-time task scheduling methods can be extended to become real-time transaction scheduling methods while concurrency control methods are still needed for operation scheduling to maintain data consistency. However, the integration of the two mechanisms in real-time database systems is not trivial, because all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: block and roll-back of transactions, both of which are barriers for time-critical scheduling. The conservative two-phase locking protocol (2PL) [Bern87] and the optimistic methods [Kung81] are examples of the two extremes. In real-time database systems, blocking can cause

† This work was supported in part by ONR N00014-88-K-0245, by IBM FSD WG-249153, and by CIT INF-90-011.

priority inversion when a high priority transaction is blocked by lower priority transactions [Sha88]. The alternative is to abort the low priority transactions if they block a high priority transaction. This implies the waste of the work done by the aborted transactions and in turn also has a negative effect on time-critical scheduling. One of the fundamental problems of real-time database systems is to develop real-time transaction scheduling protocols that maximize both concurrency and resource utilization subject to three constraints at the same time: data consistency, transaction correctness, and transaction deadlines.

We have been investigating scheduling algorithms and concurrency control protocols for real-time database systems, and implementing few of them using the real-time database prototyping tool developed at the University of Virginia [Son90], as a part of the StarLite project [Son90b]. The current version of the prototyping tool provides concurrent transaction execution facilities with various underlying synchronization mechanisms, including two-phase locking, timestamp ordering, simple priority-based contention, priority inheritance, and priority ceiling. The user can specify system configurations such as the number of sites, network topology, the number and locations of processes, the number and locations of resources, and the interaction among processes. In addition, we have been developing an experimental relational database manager for distributed real-time systems [Son90d]. In this position paper, we summarize our effort in real-time database systems research and development.

2. Concurrency Control in Real-Time Databases

The general approach to the problem of real-time transaction scheduling in most of the research work so far is to utilize the existing concurrency control protocols, especially 2PL, and apply some time-critical transaction scheduling methods to favor more urgent transactions [Sha88, Abb89, Son89]. Such approach has the inherent disadvantage of being limited by the concurrency control method upon which it is based.

Concurrency control methods rely on the setting of a serialization order among conflicting transactions. In non-real-time concurrency control methods, timing constraints are not a factor in the construction of this order. This is obviously a drawback for real-time systems. For example, with the 2PL, the serialization order is dynamically constructed and corresponds to the order in which the conflicting transactions access the shared data objects. In other words, the serialization order is bound to the past execution history with no flexibility. When a transaction T_H with a higher priority requests an exclusive lock which is being held by another transaction, T_L, with a lower priority, the only choices are either aborting T_L or letting T_H wait for T_L. Neither choice is satisfactory and thus the performance is degraded.

The situation is none the better with basic timestamp ordering protocol, since the serialization order of transactions is already determined statically by their assigned timestamps even before the execution of each transaction.

In [Abb88, Abb89], several real-time concurrency control protocols are developed by using the earliest deadline scheduling and the least slack time scheduling for a single processor database system with the two phase locking protocol for concurrency control. Though their results provide a valuable insight to the problem of real-time transaction scheduling and concurrency control, what we need is a more practical concurrency control protocol for distributed real-time database systems.

The priority ceiling protocol, which is basicly a task scheduling protocol for real-time operating systems, has been extended to the real-time database system [Sha88]. It is based on 2PL and employs only blocking, not roll-back, to solve conflicts. This makes it a very conservative approach. We have investigated methods to apply the priority ceiling protocol as a basis for real-time locking protocol in a distributed environment. One approach to implement the priority ceiling protocol in a distributed environment is to use a global ceiling manager at a specific site. In this approach, all decisions for ceiling blocking is performed by the global ceiling manager. Therefore all the information for ceiling protocol is stored at the site of the global ceiling manager.

The advantage of this approach is that the temporal consistency of the database is guaranteed, since every data object maintains most up-to-date value. While this approach ensures consistency, holding locks across the network is not very attractive. Owing to communication delay, locking across the network will only enforce the processing of a transaction using local data objects to be delayed until the access requests to the remote data objects are granted. This delay for synchronization, combined with the low degree of concurrency due to the strong restrictions of the priority ceiling protocol, is counterproductive in real-time database systems.

An alternative to the global ceiling manager approach is to have replicated copies of data objects. An up-to-date local copy is used as the primary copy, and remote copies are used as the secondary read-only copies. In this approach, we assume a single writer and multiple readers model for distributed data objects. This is a simple model that effectively models applications such as distributed tracking in which each radar station maintains its view and makes it available to other sites in the network.

We have investigated the performance characteristics of the global ceiling approach and the local ceiling approach with replication in a distributed environment. The real-time database system we have prototyped for the experiment consists of three sites with fully interconnected communication network. Our performance results have illustrated the superiority of the local ceiling approach over the global ceiling approach, at least under one representative distributed real-time database and transaction model [Son90c]. From the results of this experimentation, we believe that, even with the potential problem of temporal inconsistency (i.e., reading out of date values), the local ceiling approach is a very powerful technique for real-time concurrency control in distributed database systems.

3. New Protocols

In addition to investigating the priority ceiling protocols, we have been developing more practical real-time concurrency control protocols for distributed database systems; one is based on timestamp ordering, and the other on locking. Our approach is based on the idea of adjusting the serialization order of active transactions dynamically, by relaxing the relationship between the serialization order and the past execution history.

The first protocol schedules transactions dynamically by using their runtime estimates and deadlines to maximize both concurrency and resource utilization while satisfying database consistency and timing constraints of transactions. Furthermore, it avoids unnecessary aborting and restarting transactions by using slack time information of transactions. A transaction is characterized by its timing constraints and its data and computational requirements. The timing constraints are a release time r and a deadline d. A computational requirement is represented by a runtime estimate E which approximates the amount of computation required by the transaction. These characteristics, release time, deadline and run time estimate, are known to the scheduler when a transaction enters the system. The last characteristic, data requirements, is not known beforehand but is discovered dynamically as the transaction executes. Note that the computational requirement is simply an estimate that could be wrong or not given at all.

When a transaction is initiated, it is assigned a timestamp t by the system. The timing constraint of the transaction will be satisfied if the transaction is committed with a timestamp $t+c$, such that $0 \leq t+c \leq d-E$. We call c the *commit delay* and $t+c$ (i.e., t_c) the *commit timestamp* of the transaction. We call the range $<t, \cdots, d-E>$ the *commutability range* of the transaction and use it in our database scheduling algorithm. Any time within this range is acceptable to satisfy the timing constraint of the transaction. Once a transaction is submitted into the system, its transaction manager will be in a phase called *negotiation phase*. In this phase, the transaction manager goes into negotiation with all participating data items for a timestamp within the commutability range, at which the transaction can be committed without any conflict with other transactions. Through negotiation, the transaction manager tries to resolve the conflicts of the transaction for data consistency and to meet the timing constraint of the transaction. If, through negotiation, the system is not able to get such a timestamp within the commutability range, the transaction has to be aborted.

The negotiation is run through access requests sent to the participating data managers in the following way. To process the read or write request from a transaction, the transaction manager sends the commutability range along with the access request to the scheduler. The commutability range signifies the amount of freedom given to the data manager for scheduling the access request. If a data item is free, the data manager can schedule an access request whose timestamp is greater than the age of the data item (i.e., t_a), which is the maximum timestamp among those of the transactions whose operation is processed on the data item.

Provision of a range of timestamps gives a data manager larger maneuvering capability. If a transaction manager has provided the commutability range $<t, \cdots, d-E>$, a participating data manager can schedule the access request with timestamp $t+s$ such that $t_a < t+s \leq d-E$. We call s the *schedule delay* and $t+s$ (i.e., t_s) the *schedule timestamp* of the transaction at that data item. For some sites, the communication delay for the access request may be so large that another transaction with timestamp t_x, where $t_x > d$ has been committed ; in that case, the data manager does not accept the access request and sends a reject message to the transaction. If an access request is accepted, the data manager sends its schedule timestamp to the transaction manager. Although the access request has been scheduled at time $t+s$, the data manager knows that if the transaction succeeds, it will be committed with some timestamp (i.e., its *commit timestamp*) which lies within the range $<t+s, \cdots, d-E>$. We call this range the *commit range* of the transaction for that data item.

It is the task of transaction manager to choose a commit timestamp from a commit range. The schedule delay of access requests from the same transaction at different sites will be different. Once the commit timestamp of a transaction is determined, the transaction manager follows the two phase commit protocol.

The second protocol is a priority-dependent locking protocol, which has a flavor of both locking and optimistic approach. Our goal is to provide a locking mechanism that adjusts the serialization order, making it possible for transactions with higher priorities to be executed first so that high priority transactions are never blocked by uncommitted low priority transactions while lower priority transactions may not have to be aborted even in face of conflicting operations. For example, T_1 and T_2 are two transactions with T_1 having a higher priority. T_2 writes a data object x before T_1 reads it. In 2PL, even in the absence of any other conflicting operations between these two transactions, T_1 has to either abort T_2 or be blocked until T_2 releases the write lock. That is because the serialization order $T_2 \rightarrow T_1$ is already determined by the past execution history. T_1 can never precede T_2 in the serialization order. In our method, when such conflict occurs, the serialization order of the two transactions will be adjusted in favor of T_1, i.e. $T_1 \rightarrow T_2$, and neither is T_1 blocked nor is T_2 aborted. This priority-dependent locking protocol is free from dead-locks.

The execution of each transaction is divided into three phases: the read phase, the wait phase and the write phase. During the read phase, a transaction is executed, only reading from the database and writing to its local workspace. After it completes, it waits for its chance to commit in the wait phase. If it is committed, it switches into the write phase during which it makes all its updates permanent in the database. A transaction in any of the three phases is called an *active* transaction. If an active transaction is in the write phase, then it is committed and writing into the database.

Each lock contains the priority of the transaction holding the lock as well as other usual information. The locking protocol is based on the *principle* that high priority transactions should complete before lower priority transactions. This principle implies that if two transactions conflict, the higher priority transaction should precede the lower priority transaction in the serialization order. If a low priority transaction does complete before a high priority transaction, it is required to wait until it is sure that its commitment will not lead to the abort of a higher priority transaction. Since transactions do not write into the database during the read phase, write-write conflicts need not be considered here. With our new time-critical transaction scheduling policy, a high priority transaction will commit before a low priority transaction most of the time [Lin90].

References

[Abb88] R. Abbott and H. Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," *Proceedings of the 14th VLDB Conference*, 1988.

[Abb89] R. Abbott and H. Garcia-Molina, "Scheduling Real-time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[Bern87] P. Bernstein, V. Hadzilakos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.

[Kung81] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems 6*, 2, June 1981.

[Lin90] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," submitted for publication.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, Vol. 17, No. 1, March 1988.

[Son88] S. H. Son, "Real-time Database Systems : Issues and Approaches," *ACM SIGMOD Record 17*, 1, March 1988.

[Son89] S. H. Son, "On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems," *IFAC/IFIP Workshop on Distributed Databases in Real-Time Control*, Budapest, Hungary, October 1989.

[Son90] S. H. Son, "An Environment for Prototyping Real-time Distributed Databases," *International Conference on Systems Integration*, April 1990.

[Son90b] S. H. Son and R. Cook, "StarLite: An Environment for Prototyping and Integrated Design of Distributed Real-Time Software," *Second International Conference on Computer Integrated Manufacturing*, May 1990.

[Son90c] S. H. Son and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, June 1990.

[Son90d] S. H. Son and M. Poris, "An Experimental Database Manager for Distributed Real-Time Systems," submitted for publication.

[Stan88] J. Stankovic, "Real-time Computing Systems : The Next Generation," *Technical Report 88-06*, Department of Computer and Information Science, Univ. of Massachusetts.

Preemption *vs.* Priority, and the Importance of Early Blocking

T.P. Baker*
Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

March 31, 1990

Abstract

The Stack Resource Policy, which was developed to allow processes to share a single runtime stack with bounded priority inversion and no deadlock, illustrates the advantages of early blocking and distinguishing "preemption level" from priority – two principles which appear to have broader applications.

Introduction. The importance of bounding priority inversion in priority-driven real-time scheduling has been demonstrated by Sha, Rajkumar, and Lehoczky[6]. Extending the work of Liu and Layland[4], they obtained simple *a priori* schedulability tests for systems of periodic tasks under rate-monotone scheduling, even when those tasks have critical sections protected by binary semaphores. Their results are based on following a policy that insures that for each process $\mathcal{P}_i$ there is a constant B_i that bounds the length of time any job of $\mathcal{P}_i$ may be subject to priority inversion. In [6], they propose several semaphore-locking policies with this bounded priority inversion property, including the Priority Ceiling Protocol (PCP). In [5], they describe another such policy, called the Semaphore Control Protocol (SCP), which is optimal in the sense of imposing no more blocking than is necessary to bound priority inversion and avoid deadlock.

Resource Modes, and the MBP. Extending the resource model of [6] to include allocation *modes*, we have shown in [1] that bounded priority inversion policies can be defined for more general resource allocation problems, including reader-writer resources. We defined a policy for general resources, called the Minimal Blocking Policy (MBP), and proved that it is optimal in the same sense as the SCP. Although the MBP is a consistent generalization of the SCP, it is defined differently. The MBP is defined via a least-fixed-point solution to a recurrence that forbids deadlock and multiple priority inversion, without reference to priority ceilings. The recursive definition solves a problem with circularity encountered by the definition of the SCP given in [5], and the choice of least fixed-point makes the optimality property of the MBP follow more directly.

Stack Sharing. In [2], we reconsider the issue of priority inversion under the assumption of a shared runtime stack. The original motivation for this application was a system with very many processes, but only a few distinct priority levels. The processes are mixed periodic and sporadic. They are to be executed on a very fast microprocessor with a large on-chip state, so that context switches are relatively expensive. Since only a few processes can be active at any one time, memory allocated for the runtime stacks of the waiting processes is under-utilized. Therefore, in order to reduce the total system memory requirement, it is desirable to have processes share one runtime stack. The stack sharing scheme is illustrated in Figure 1.

*This work supported in part by grant N00014-87-J-1166 from the U.S. Office of Naval Research. This paper submitted to the Seventh IEEE Workshop on Real-Time Operating Systems and Software.

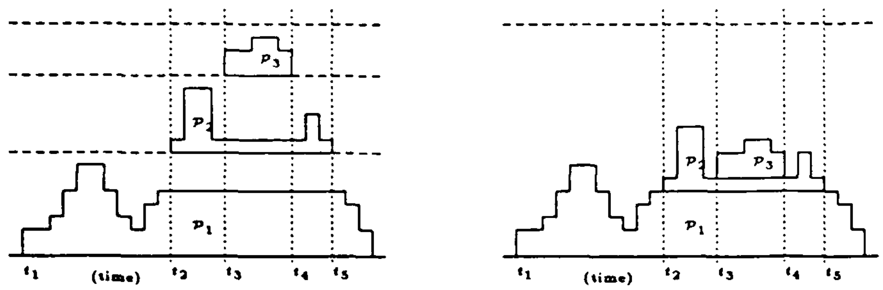


Figure 1: Stack per process *vs.* stack sharing.

The critical constraint imposed by the shared stack is that once a process is preempted it cannot be resumed until the process that preempted it completes (and releases the stack space immediately above the preempted process). Without precautions, this easily leads to deadlock, so it is essential to include the runtime stack among the nonpreemptable resources governed by a policy such as the PCP. Applying the principles of the PCP and MBP under the assumption of a shared runtime stack, we obtain two new policies, which we call the Stack Resource Policy (SRP), and the Minimal Stack Resource Policy (MSRP).

These turn out to be simpler to implement than the PCP and the SCP/MBP. They also turn out to be an improvement in two other respects:

1. They apply directly to deadline-driven scheduling, as well as rate-monotone scheduling.
2. They eliminate the possibility of one wasted context switch per preemption.

These improvements make the SRP and MSRP arguably superior to the PCP, even for situations where there is no stack sharing. Moreover, retrospective examination of these policies reveals that these improvements are due to tw simple observations which may have broader applications. In order to explain these observations, and the stack resource policies themselves, we need to define our model of a real-time system.

The Model. In our model, a real-time system is composed of a number of *processes*, which in turn are are composed of a finite number of *jobs*. A job is a finite sequence of instructions. This sequence of instructions may include conditional control flow, but must execute in bounded time. The instructions may include request and release operations for certain *nonpreemptible resources*.

Each resource may be requested in a finite number of *modes*. For example, a shared-data resource would have two modes, *read* and *write*, while a binary semaphore or unit of stack space would have only one mode, equivalent to *write*.

A resource *allocation* A is a triple (J, R, m), where J is a job, R is a resource, and m is a mode. A job starts and completes execution holding no allocations, and must release allocations in LIFO order. The granting of resource allocation requests is governed by a *resource allocation policy*. This policy is constrained to at least block those requests for which there are already conflicting usages outstanding, as represented abstractly by a binary relation on allocations, called the *direct blocking* relation. (For example, any allocation of a binary semaphore directly blocks all requests for that semaphore by other jobs.) In order to avoid priority inversions, the policy may also block some other requests, which are not directly blocked.

Each process determines a sequences of *job-execution requests*. Each job-execution request $\mathcal{J}$ specifies a *job*, J, an *arrival time*, $Arrival(\mathcal{J})$, and a *priority*, $p(\mathcal{J})$. The priority is fixed at the time the request arrives, but may vary from request to request for the same job, and within the same process. For each job-execution request, the job is executed once, from beginning to end. The job-execution requests of each process are processed sequentially, so that executions of the same job never overlap. The processor is allocated to job executions preemptively according to priority, and FIFO within priority classes, with priority inheritance through any held resources.

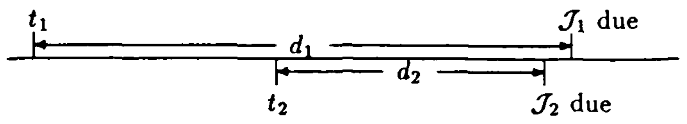


Figure 2: Preemption, in deadline-driven scheduling.

Each job J has a fixed *preemption level*, $\pi(J)$, which is a positive integer. Preemption levels are related to arrival times and priorities by the condition:

$$p(\mathcal{J}) \leq p(\mathcal{J}') \lor Arrival(\mathcal{J}) \leq Arrival(\mathcal{J}') \lor \pi(J') < \pi(J).$$

Intuitively, this says that $\pi(J') < \pi(J)$ if $\mathcal{J}'$ can be preempted by $\mathcal{J}$, and is based on the observation that for $\mathcal{J}$ to actually preempt $\mathcal{J}'$, it must have higher priority than $\mathcal{J}'$ and arrive after $\mathcal{J}'$ has started to execute.

Preemption Levels *vs.* Priorities. The preemption level of a job may, but need not, be equal to the priority of requests for that job. They are equal, for example, when we apply our model to rate-monotone scheduling. Each process has a unique priority, and all the job-execution requests of the process have that priority. If we require that each job be requested by only one process, we can let the preemption level of each job be the priority of the process that requests it, which is the same as the priority of the requests for that job. In contrast, for deadline-driven scheduling there is no such simple relationship between preemption level and priority.

In deadline-driven scheduling, we assume each job J has a fixed *relative deadline*, d_J. If a request for execution of J arrives at time t, that execution must be completed by time $t + d_J$. The relative deadline of a job is the size of the scheduling "window" in which each execution of the job must fit.

Suppose there are two jobs J_1 and J_2, with relative deadlines d_1 and d_2, respectively. Suppose $\mathcal{J}_1$ is a job-execution request of J_1 that arrives at time t_1, and $\mathcal{J}_2$ is a job-execution request of J_2 that arrives at time t_2. In order for $\mathcal{J}_2$ to preempt $\mathcal{J}_1$, we must have the situation shown in Figure 2, that is:

i. $t_1 < t_2$ (so $\mathcal{J}_1$ can get started);
ii. $t_1 + d_1 > t_2 + d_2$ (so $\mathcal{J}_2$ can preempt).

It follows from conditions (i) and (ii) above that $d_1 > d_2$. Thus, we know that *a job can only preempt another job with a shorter relative deadline.* This means we can assign preemption levels to jobs according to their relative deadlines. In particular the preemption level of a job J may be defined to be $0 - d_J$.

Preemption Ceilings. We can now generalize the idea of "priority ceiling" in [6], by defining the *preemption ceiling*, $\lceil A \rceil$, of a resource allocation A to be the maximum of zero and the preemption levels of all the jobs that may be blocked directly by A. Like the priority ceiling, the preemption ceiling of a resource allocation may be statically determined.[1]

At any instant of time, let the *current ceiling*, $\overline{\pi}$, be the maximum of the preemption level of the currently executing job and the preemption ceilings of all the outstanding allocations. That is, if job J is currently executing,

$$\overline{\pi} = max(\{\pi(J)\} \cup \{\lceil A' \rceil \mid A' \text{outstanding}\}).$$

Stack Resource Policies. The SRP blocks a job-execution request $\mathcal{J}$ from from starting execution, until $\overline{\pi} < \pi(J)$. Once a job J has started execution, all subsequent resource requests by J are granted immediately, without blocking.

[1]This is unlike the "dynamic priority ceilings" of [3].

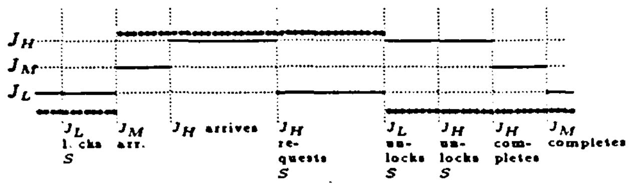


Figure 3: Execution with PCP.

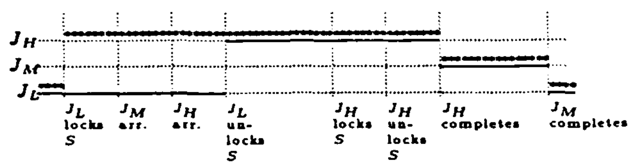


Figure 4: Execution with SRP.

The MSRP weakens the condition for a job to be allowed to start. Let J_C be the job that is currently executing, if any, and A_C be the resource allocation currently held by J_C that has highest ceiling, if J_C is holding any non-stack allocation. The MSRP requires that a job execution request $\mathcal{J}$ be denied an initial stack allocation A (i.e. not occupy stack space above J_C) iff there is an outstanding allocation A' such that either of the following conditions is satisfied:

1. $\pi(J) < \overline{\pi}$;
2. if $\pi(J) = \overline{\pi}$ and J may request an allocation A_2 that is directly blocked by A_C.

Once J has started execution, all subsequent resource requests by J are granted immediately, without blocking.

Note that the blocking tests are only applied before a job starts. However, this does *not* mean that the all the resources that may ever be requested by J are locked at that time. They are only allocated when requested, and are released when they are no longer needed. Thus, if J later requests some allocation A and there is a higher level job J_H that will need some allocation A_H that is blocked by A, J_H is free to J preempt so long as J is not in a critical section for A.

It is proven in [2] that the SRP and MRSP guarantee no job is subject to priority inversion for longer than the duration of one (outermost) critical section of another job. Moreover, the MSRP imposes the minimum blocking necessary to guarantee this bounded priority inversion property. Even if there is no stack sharing, the SRP is at least as good as the PCP in reducing worst-case priority inversion.

Early Blocking. Another interesting property of the SRP and MSRP is that they reduce the worst-case number of context switches to one per job-execution request, as compared to the PCP and SCP, through earlier blocking. Resource allocation policies, such as the PCP, that avoid multiple priority inversion but allow late blocking permit the following scenario:

1. A high priority job, J_H preempts the processor from a lower priority job, J_M, and executes for a while.
2. J_H is forced to allow a lower priority job J_L (preempted earlier by J_M) to resume execution, because J_L is holding a resource needed by J_H.
3. J_L releases the resource needed by J_H, and J_H resumes execution.
4. J_H completes, and J_M resumes execution.

Thus, the execution of J_H may require *four* context switches. This is illustrated in Figure 3. The solid horizontal lines indicate which job is executing. The horizontal lines of asterisks indicate the relative value of $c(S^*)$, the current ceiling under the PCP. (Note that $c(S^*)$ may be less than $\overline{\pi}$, since it does not include the resources held by the current job. but this does not make any practical difference.)

By comparison, the worst-case scenario permitted by the SRP has only *two* context switches, as illustrated in Figure 4:

1. J_H waits until $\pi(J_H) > \overline{\pi}$, then begins execution, preempting J_L. (Note that J_M and J_H are forced to wait until J_L releases S, since $\overline{\pi} \geq \pi(J_H)$.)
2. J_H completes and J_M resumes execution.

Conclusion. We have explained three basic ways to improve on the family of semaphore allocation policies developed by Sha, Rajkumar, and Lehoczky. These are:

1. *Modes.* By introducing resource allocation modes, one can handle a wider variety of nonpreemptible resources.
2. *Preemption levels.* By separating the concept of *preemption level* from priority, one can accomodate some dynamic priority schemes, including Liu and Layland's deadline-driven scheduling.
3. *Early blocking.* By blocking a job before it starts, in the case that one can predict that the job will be blocked at some point during its execution, a wasteful context switch can be avoided and the overhead of checking the blocking condition for every resource request can be eliminated.

These ideas lead to several interesting questions, some of which we are currently pursuing.

It appears that the principle of separating preemption from priority may permit extending more of the known refinements of Liu and Layland's work on rate-monotone scheduling, to similar results for deadline-driven scheduling. The principle of early blocking also seems to have a wider application. Though we have applied it to the PCP and SCP/MBP, it is also applicable to other resources allocation policies. In fact, we can show that if the resource requests of every job can be predicted unconditionally, early blocking never lengthens the response time of any job without shortening the response time of a higher priority job.

We are also able to make a slight further generalization of the resource model, to permit the treatment of multiunit resources that allow requests for "m out of n" units. We are currently working on a detailed description of this generalization as it applies to the MBP. ......resume here ..... The key idea is to extend the definition of blocking to allow more complex forms of blocking, by sets of allocations rather than individual allocations. Since any policy that insures bounded blocking will not permit the most complex forms of blocking, the implementation can be more efficient than would otherwise be expected.

References

[1] T.P. Baker, "A Fixed-Point Approach to Bounding Blocking Time in Real-Time Systems", technical report, Department of Computer Science, Florida State University, Tallahassee, FL 32306 (July 1989).

[2] T.P. Baker, "Stack-Based Scheduling of Realtime Processes", technical report, Department of Computer Science, Florida State University, Tallahassee, FL 32306 (December 1989).

[3] M.I. Chen and K.J. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems", technical report UIUCDCS-R-89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign (April 1989).

[4] C.L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", JACM 20.1 (January 1973) 46-61.

[5] R. Rajkumar, L. Sha, and J.P. Lehoczky, "An Optimal Priority Inheritance Protocol for Real-Time Synchronization", technical report, Carnegie Mellon University (17 October 1988) submitted for publication.

[6] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", technical report CMU-CS-87-181, Carnegie Mellon University (November 1987).

Supporting Real-Time Concurrency [1]

Victor Wolfe, Susan Davidson, and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

1 Introduction

Real-time applications such as robotics, industrial control and avionics, frequently operate in distributed environments that require complex concurrency control under timing constraints. However, it is currently unclear what basic concepts are needed for expressing concurrency requirements in the presence of timing constraints. For example, consider a robotics application where two robot arms must lift a container of chemicals from a moving conveyer belt. The arms are shared among the lifting task and other tasks that execute concurrently in the application. To prevent spills when lifting, the following constraints on the operation of the arms must be satisfied: the arms should lift simultaneously, no other use of the arms should be allowed while the lift is being performed, and either both arms should lift or neither arm should lift. Furthermore, the lifting should be scheduled to meet timing constraints that arise from the dynamics of the moving belt and inherent properties of robot control algorithms.

To support such distributed real-time applications, a programming language and run-time support system should have the following characteristics. First, the development of *correct* programs should be facilitated. A correct program meets all of the functional and timing constraints imposed by the application. Many of these constraints are illustrated in the example; they include serializability for resources, precedence ordering, absolute timing constraints, exclusive execution, simultaneous execution, and all-or-nothing execution. Second, to improve resource utilization and support the enforcement of timing constraints, *concurrent* access to resources should be allowed whenever it can be ensured that functional constraints will not be violated. Third, to facilitate the prediction of timing behavior, *priority blocking* should be avoided whenever possible. Priority blocking occurs when a lower-priority process keeps a higher-priority process from executing by holding resources required by the higher-priority process. Fourth, to facilitate programming, *modularity* and *abstraction* should be supported in the expression and enforcement of correctness constraints.

[1]This work is support in part by the following grants: ONR N000014-89-J-1131, ARO DAAG-29-84-k-0061, NSF CCR87-16975 and a grant from AT&T Telecommunications Program at the University of Pennsylvania

Current techniques for supporting concurrency control in real-time systems are mainly concerned with scheduling of CPU and not of other shared resources such as memory and devices. For instance, Modula-2 [1], Real-Time Euclid [2], and Ada [3] control access to these shared resources using mutual exclusion techniques (*monitors* in [1,2] and *rendezvous* in [3]). There are several problems with the use of these mutual exclusion techniques in real-time systems. Firstly, they deny potential concurrency that can be valuable in meeting timing constraints. Secondly, because no preemption of resources is allowed in mutual exclusion techniques, and because (non-CPU) resources are typically scheduled first-come-first-serve, priority blocking may occur. Finally, they do not ensure simultaneous use of different resources or all-or-nothing performance of a set of actions on the resources.

In this paper, we present a basic set of necessary concepts for high-level expression of timing constraints and concurrency requirements in a distributed real-time program. Our discussion is based on an abstract data-type model with transactions [4,5]. Our model consists of *resources* and *processes*. Resources provide the abstract views of system entities, such as devices and memory, which can be shared. Each resource defines a set of actions that can be invoked from outside and also specifies permissible interleaving of the executions of actions. Processes manipulate resources by invoking actions. A process specifies an ordering and timing constraints for executing actions; in addition, it also specifies concurrency requirements on a subset of actions, such as requiring that they be *exclusive*, *simultaneous*, or *atomic*. We conclude by briefly describing our current work in developing language constructs and run-time support to implement the concepts presented in this paper.

2 Resources

All components of the system that can be shared, such as CPU, memory, communication networks, and devices, can be defined as *resources*. A resource, r, is characterized as $\langle S_r, A_r, C_r \rangle$. S_r is a set of states. In the example, a robot arm is a resource; its state represents the Cartesian coordinates of the arm and the position of the hand (grasped/ungrasped). A_r is a set of *actions*. A process invokes an action as the only means to use the resource. The action invocation changes the resource state and returns values to the process. In the example, each robot arm's actions include: *lift, lower, grasp*, and *read* (which returns the position of the robot arm). C_r is a *compatibility predicate* that describes conflict between actions such that, for two actions $a_1 \in A_r$ and $a_2 \in A_r$, $C_r(a_1, a_2) = true$ if all interleavings of the execution of a_1 and a_2 produce the same state for r and the same return values for a_1 and a_2. For instance, $C_{arm}(lift, grasp) = TRUE$ because the actions operate on different parts of the resource state; $C_{arm}(read, read) = TRUE$ because *read* does not change the state and they return the same values in any interleaving;

$C_{arm}(lift, lower) = FALSE$ because the actions change the state in conflicting ways.

A *schedule* of a resource r is $Sch_r = \langle H_r, start_r, complete_r \rangle$, where H_r is a set of all invocations of actions on the resource; $start_r : H_r \rightarrow time$ maps each action invocation in H_r to the absolute time which it started executing; and $complete_r : H_r \rightarrow time$ maps each action invocation in H_r to the absolute time which the action terminated. A resource's schedule defines a partial order, $\prec_r$, on the action invocations of H_r such that, for two invocations $a_1 \in H_r$ and $a_2 \in H_r$, $a_1 \prec_r a_2 \Rightarrow complete_r(a_1) \leq start_r(a_2)$. The ordering $\prec_r$ is partial because the execution of actions may be interleaved in the schedule and thus neither $complete_r(a_1) \leq start_r(a_2)$ nor $complete_r(a_2) \leq start_r(a_1)$.

A *functional constraint* of a resource is a predicate on the values of the state variables of the resource. Each resource has a set of functional constraints that must be maintained to be correct. These functional constraints are the inherent properties of its design. For instance, a robot arm may be constrained to have positive Cartesian coordinates that are bounded by maximum values. We assume that a resource is designed such that each action preserves the functional constraints of the resource if its execution is not interleaved with the execution of an incompatible action of the same resource.

From this assumption, we can derive a sufficient scheduling constraint for guaranteeing the functional constraints of a resource. We say that a schedule for a resource r is *serial* iff $\prec_r$ is a total order on the action invocations of H_r; *i.e.*, there is no interleaving of action invocations. For a given resource r, two schedules Sch_1 and Sch_2 are *equivalent* iff $H_1 = H_2$, the resource has the same final state in Sch_1 and Sch_2, and every action invocation of H_1 has the same return value in Sch_1 as the corresponding action invocation in Sch_2. A schedule is called *serializable* iff it is equivalent to a serial schedule. Therefore, if the schedule of a resource is serializable, then the functional constraints of the resource are maintained.

3 Processes

A process consists of action invocations and constraints on the scheduling of those invocations. In particular, a process P is defined as $\langle AI_p, \prec_p, SYNCS_p, SIGS_p, TS_p, ES_p, SS_p, AS_p \rangle$. AI_p is a set of action invocations. $\prec_p$ is an irreflexive partial ordering on AI_p such that if $a_1 \prec_p a_2$, the execution of a_1 must be completed before a_2 can start. If $a_1, a_2 \in A$ are not related by $\prec_p$, then they may be executed concurrently. In the robot example, the ordering of action invocations is as follows:

$$\begin{tikzcd}
\mathtt{read}_{arm1} \arrow[r] \arrow[dr] & \mathtt{grasp}_{arm1} \arrow[r] \arrow[dr] & \mathtt{lift}_{arm1} \\
\mathtt{read}_{arm2} \arrow[r] \arrow[ur] & \mathtt{grasp}_{arm2} \arrow[r] \arrow[ur] & \mathtt{lift}_{arm2}
\end{tikzcd}$$

Unfortunately, expressing action precedence using only local orderings in processes is not sufficient in most concurrent systems where processes interact. In these systems, precedence orderings among action invocations in different processes must be provided for *interprocess synchronization.* Interprocess synchronization requirements are expressed by two sets: a *sync set*, $SYNCS_p$, and a *signal set*, $SIGS_p$. An action invocation $a_i \in AI_p$ is a *sync action*, $a_i \in SYNCS_p$, iff it is the first action invocation of a process to be ordered after a particular action invocation in another process. An action invocation $a_i \in AI_p$ is a *signal action*, $a_i \in SIGS_p$, iff it is the last action invocation of a process to be ordered before a particular action invocation in another process. Each of a process's sync actions must be scheduled after all of its corresponding signal actions have completed.

TS_p is a set of timing constraints on action invocations. For instance, P_{lift} has a deadline that must be met to adhere to the dynamics of the conveyer belt. Each element $ts \in TS_p$ is called a *temporal scope* and defined as $ts = \langle T, sa, sb, d\rangle$, where sa is an absolute earliest start time, sb is an absolute latest start time, d is an absolute deadline, and T is the set of action invocations to be time constrained. T has the following property: if $a_1, a_2 \in T$, then for every $a \in AI_p$ such that $a_1 \prec_p a$ and $a \prec_p a_2$ implies $a \in T$. Note that periodic behavior can be expressed by a series of temporal scopes where for each temporal scope, ts_i, of the series: $d_i - sa_i =$ the period, and $sa_i = d_{i-1}$.

The temporal scopes of a process are either nested or disjoint; that is, for all $ts_1, ts_2 \in TS_p$, either $T_1 \cap T_2 = \emptyset$ or $T_1 \subseteq T_2 \vee T_2 \subseteq T_1$. A process may nest temporal scopes to impose tighter timing constraints. If a temporal scope ts_2 is *nested* in another temporal scope ts_1, ts_2 *inherits* the timing constraints of ts_1 as follows: The initial temporal scope for the process is $\langle AI_p, 0, \infty, \infty\rangle$. When a temporal scope $ts_2 = \langle T_2, sa_2, sb_2, d_2\rangle$ is nested in the current temporal scope $ts_1 = \langle T_1, sa_1, sb_1, d_1\rangle$, the actual temporal scope $ts'_2 = \langle T', sa'_2, sb'_2, d'_2\rangle$ must satisfy these requirements: $sa'_2 \le sb'_2 \le d'_2$; $d'_2 \le d_1$; and $d'_2 \le d_2$. To satisfy these requirements, the actual temporal scope for T_2 is computed at run-time as follows: $sa'_2 := sa_2$, $d'_2 := \min(d_1, d_2)$, and $sb'_2 := \min(sb_2, d'_2)$. For an action invocation, its initial temporal scope is the current temporal scope of the calling process when the action is called.

4 Concurrency Requirements

In addition to satisfying a partial execution order and a set of timing constraints, the correct execution of a process may require the exclusive use of resources (ES_p), the simultaneous use of multiple resources (SS_p) and the atomic use of multiple resources (AS_p).

A process may require that a sequence of actions on the same resource be performed without interference from another process. For instance, on each robot arm P_{lift} must perform *read*, *grasp*, and *lift* actions without another process executing an action that moves the arm. Each exclusive set es in ES_p of a process identifies a set of action invocations on a single resource that must be executed without another process executing incompatible actions. To ensure the exclusive access during the execution of actions specified in an exclusive set, every exclusive set es must satisfy the following property: if $a_1, a_2 \in es$, then for every $a \in AI_p$ such that $a_1 \prec_p a$, $a \prec_p a_2$ and a, a_1, a_2 are action invocations on the same resource implies $a \in es$. Furthermore, every action of a process is in exactly one exclusive set (possibly a single-action exclusive set). Given a set of exclusive sets, a scheduling of a resource satisfies the exclusive set requirement if it is equivalent to a schedule in which no actions from different exclusive sets are interleaved. In the example, P_{lift} has two exclusive sets: $ES_{Plift} = \{\{read_{arm1}, grasp_{arm1}, lift_{arm1}\}, \{read_{arm2}, grasp_{arm2}, lift_{arm2}\}\}$.

In real-time systems, there may be sets of actions that must be executed at the same time. The partial order $\prec_p$ constrains the relative sequencing of action invocations and allows a set of actions to be executed concurrently, but makes no provision for requiring *simultaneous execution* of actions. For instance, although the two *lift* actions are concurrent in P_{lift}, one *lift* action may complete just as the other is starting, causing the container to spill. To express the requirement of simultaneous execution, SS_p specifies sets of action invocations that must start executing at the same time and must not be preempted. In the example, P_{lift} has a simultaneous set: $SS_{Plift} = \{\{lift_{arm1}, lift_{arm2}\}\}$.

A final constraint that may be posed by a process is that a set of action invocations either all must be performed or none must be. For instance, P_{lift} should perform a *lift* action on one arm iff the other *lift* action is also performed. Such a set of action invocations is called an *atomic set*; AS_p is the set of atomic sets of a process. P_{lift} has a single atomic set: $AS_{Plift} = \{\{lift_{arm1}, lift_{arm2}\}\}$.

5 Current Work

We are currently developing language constructs for the concepts presented in this paper. Serializability constraints on resource schedules are met by restricting scheduling so that interleaving is only among compatible actions [6]. Timing constraints are expressed using *temporal scope* language constructs [7], which also provide exception handling for violated timing constraints. Exclusive sets and simultaneous sets are explicitly expressed using *exclusive block* and *simultaneous block* constructs, respectively [6]. Atomic sets are supported by language constructs for *timed atomic commitment* [8,9]. We plan to implement these language constructs and their run-time support using a real-time kernel also being developed at the University of Pennsylvania [10].

References

[1] N. Wirth, *Programming in Modula-2*. New York: Springer-Verlag, 1983.

[2] E. Klingerman and A. Stoyenko, "Real-time Euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 941–949, Sep. 1986.

[3] U.S. Department of Defense, "Ada Programming Language," 1983. ANSI/MIL-STD-1815A-1983.

[4] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, pp. 300–312, March 1988.

[5] T. Hadzilacos and V. Hadzilacos, "Transaction synchronization in object bases," in *ACM Priciples of Distributed Systems Conference*, pp. 193–200, 1988.

[6] V. Wolfe, S. Davidson, and I. Lee, "Language constructs for distributed real-time consistency," Tech. Rep. CIS-89-82, Department of Computer and Information Science, University of Pennsylvania, 1989.

[7] I. Lee and V. Gehlot, "Language constructs for distributed real-time programming," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1985.

[8] S. Davidson, I. Lee, and V. Wolfe, "Language constructs for timed atomic commitement," in *19th International Symposium on Fault-Tolerant Computing*, June 1989.

[9] S. Davidson, I. Lee, and V. Wolfe, "Timed atomic commitment," Tech. Rep. MS-CIS-88-80, Department of Computer and Information Science, University of Pennsylvania, Oct. 1988. To appear in *IEEE Transactions on Computers*.

[10] I. Lee, R. B. King, and R. P. Paul, "A predictable real-time kernel for distributed multi-sensor systems," *IEEE Computer*, vol. 22, pp. 78–83, June 1989.

RTL meets ORE

Marc Donner and Farnam Jahanian
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

1 Introduction

This position paper describes some research into the adaptation of RTL, a formalism for specification of and reasoning about temporal properties of real-time programs, to use in the ORE real-time systems programming language. The reasons for this collaboration are many, but most basic among them is the fact that ORE needed tools for expressing temporal ideas and RTL is both appropriate to the task and intuitively appealing.

2 Events, occurrence functions, RTL formulae

RTL [4] is a formalism for specifying and reasoning about the real-time properties of systems. RTL views a computation of a real-time system as a sequence of event occurrences. Informally, events represent things that can happen in a system. An event occurrence, however, defines a point in time at which a particular occurrence of an event happens in a computation. Hence, a timing property of a system, such as periodic execution or a deadline on completion of a task, can be specified as a relationship among event occurrences. The execution of an action is modeled by two events: one denoting its initiation and the other marking its completion. Changes in the state variables in a system are denoted by transition events.

The notion of real-time is captured by the occurrence function, denoted by '@'. The occurrence function assigns a time value to each occurrence of an event. Informally, @(**e**, **i**) is the time of the ith occurrence of event **e**. An RTL formula is constructed using the occurrence function, the relations $=, <, \leq, >$, and $\geq$, the first-order logic connectives $\neg, \wedge, \vee$, and $\rightarrow$, and the $\forall$ and $\exists$ quantifiers. The following is an example of an RTL formula:

$$\forall i \ @(e, i) + 100 \leq @(e, i + 1)$$

which states that there is a minimum separation of 100 time units between consecutive occurrences of event **e**.

3 ORE events

In ORE there are two classes of events that are observable. The first class corresponds to the start and stop events of actions in RTL. These are defined in an ORE program by inserting labels in appropriate places in the code.

```
...
E1->
S;
<-E2
...
```

In the example above, two event labels are defined, **E1**, and **E2**. The right-pointing-arrow is a syntactic marker that specifies that E1 is the event that denotes the start of S. The left-pointing-arrow tells that E2 marks the end of S. Note that it may be useful in some situations to have two events between a pair of statements, one bound to the end of the first and one bound to the beginning of the second. Preemption between two statements is observable in an execution.

The second class of event corresponds roughly to the transition events in RTL. What causes the ORE event is assignment to a *watchable* variable. A watchable variable is one that has been specially declared to have the property that assignment *to it is observable* asynchronously. In a certain sense, the ORE watchable variable is really an overloading of one name with two things. The name is a state variable of the language, which may be assigned and used in expressions, and it is an event that can be observed with the asynchronous event observers described in the next section.

If the state of a program can be completely described by the value of its program counter and the values of all of its state variables, then the two kinds of ORE events correspond to program counter assignment in one case and to state variable assignment in the other. In this sense, no significant component of an ORE program is inaccessible to observation as an event.

4 Watch and when

Watch and when are ORE language elements that permit asynchronous observation of event occurrences by ORE processes. The most primitive element is **watch** which suspends a thread of control waiting for any one of a list of named events to occur. When one of the events named has occurred, then the thread resumes. There is an optional **do** clause for the **watch** statement that permits a single statement or strip to be associated with the awakening of the strip in an indivisible way. A watch statement looks like this:

watch v_1, v_2, ..., v_n do S;

When is a more powerful primitive that permits a thread of control to suspend while a Boolean-valued expression is false, resuming when it becomes true. There is a **do** clause for **when** as well, though it has a somewhat stronger property. The code strip that is introduced by the **do** clause of a **when** statement is guaranteed that when it is executed the Boolean expression that it depends on will be true. As a result, the **when** statement is a high-level *test-and-set* statement. A when statement looks like this:

```
when B do S;
```

The **when** statement might be implemented in terms of the more primitive **watch** statement:

```
<
{ if B then { S; break; }; };
watch dependent-list( B );
>
```

where dependent-list is a function that returns a list of the watchable variables upon which BooleanExpression depends. Notice that if between the awakening of the **watch** and the execution of the **do** clause something causes the BooleanExpression to become false again, the **if** will fail and the **while** loop will reiterate.

5 Event occurrences: history and data structures

The initial design of ORE focussed exclusively on concurrent programming and asynchronous event notification features [1,2]. Our concern for tools for talking about time in ORE programs led us toward certain RTL-like features, in particular a notion of associating the time of occurrence with each event. This led, after study of the RTL work, to the decision to include a finite history into ORE watchable variables. The history would capture a number of previous assignments to the watchable variable, recording both the value and the time of assignment. With this information available a large number of RTL formulae would be available at runtime to the ORE programmer.

Here is an example, in C, of what the data structure for a watchable variable of type **watchable_foo**, a primitive type representable in a single word of storage, would look like. In the example, the number of past values of one of these variables that is kept is defined in the constant **H**, which is a per-variable value.

```
struct w_foo
   {
   process *foo_watchers;
   int foo_index;
   time foo_time[ H ];
   foo foo_value[ H ];
   } watchable_foo;
```

When an assignment is to a variable of type **watchable_foo** is encountered in the ORE source code, the following C code might be generated when assigning an expression to a watchable variable x:

```
/* "x = exp;" is the ORE code */
var ix :  int;
ix = ( x.foo_index + 1 ) % H;
x.foo_value[ ix ] = exp;
x.foo_time[ ix ] = now();
x.foo_index++;
do_wakeup( x.foo_watchers );
```

It must be noted that the integer variable **foo_index** is the absolute occurrence index of the current (last) event in the history. Since the finite history is implemented as a circular list in the above code segment, the modulus operator is used to calculate the position within the circular list. An alternative formulation allows implementation of an assignment to a watchable variable without requiring the modulus operator. The cost is one extra integer variable in the structure **watchable_foo** to keep the current position in the circular history list, i.e., absolute index of current occurrence modula **H**.

ORE provides an RTL-like notation for accessing both the times of assignment and the values of historical events. The time of a previous occurrence of an event is named by `@( event_name, -a )` where a is a constant between 1 and H. `@( event, -1 )` is the most recent time at which *event* has occurred, and `@( event, -H )` is the least recent still remembered instant. Of course, since only a finite number of occurrences of each event are remembered, references out of bounds must be handled appropriately. To avoid subtle aliasing problems with `@(event_name, -a)`, we can also allow the notation `@(event_name, i)` such that `i` is a program variable. The time of the ith occurrence of the event is obtained from the history. Again, references out of bounds must be handled. The latter notation allows manipulation of event histories and occurrence indices via ORE language constructs. For example, all remembered occurrences of an event can be examined in a loop construct.

In RTL state variables are related to events by a Boolean test, so that there can be an event for x being equal to 7 and a different event for x being greater than 7. In ORE all assignments to x are events and the value is not distinguishable in that way. In order to provide access to historical values of a watchable variable, ORE provides the `@val` operator. `@val( x, -a )` is the a'th oldest value that was assigned to x. `@val( x, -1 )` is always the most recent value and is synonymous with `x`.

6 Runtime RTL

The primary advantage of adapting RTL to specifying runtime properties of real-time systems is that RTL distinguishes between different occurrences of the same events. However, adding RTL-like constructs to a programming language raises a number of issues. Among them are questions of reducability of RTL formulae to algorithms, finiteness of event histories, syntax of specifications, and interaction between language features and RTL formulae. In this section we will discuss two different approaches that we are considering. Our initial thought is that both approaches are needed.

6.1 Profane RTL - the implementor's view

ORE events and watchable variables, along with the occurrence functions `@( e, -a )` and `@val( e, -a )` reduce the corresponding RTL notions to program state variables. As a result, they may be used almost anywhere in an ORE program. The only tricky detail is that, for instance, occurrence functions are themselves watchable and produce wakeups in **watch** and **when** statments whenever the underlying event occurs.

By adding RTL-like occurrence functions to ORE one can now express things that are difficult to specify in RTL because it is not intended for functional specification. An example of this is deadline satisfaction in a client-server model. In ORE using watchable variables this can be implemented as follows:

```
var
   req :  watchable int;
   ack :  watchable int;
proc server
   <
   when req != 0;   /* wait for a request */
   .../* handle the request */
   when ack = 0 do { ack = req; req = 0; }
   >
proc client( id :  int )
   <
   when req = 0 do req = id;   /* wait for idle server */
   when ack = id;   /* we have been served */
   .../* gather up the result */
   ack = 0;   /* clear the acknowledge */
   .../* Do other work */    >
[
   server;
   client( 1 );
   client( 2 );
   client( 3 );
   client( 4 );
]
```

We would like to ensure that each client is served promptly after its request. We can do that by verifying that each occurrence of **ack** happens within deadline **d** of its **req**. This requirement is slightly tricky with pure RTL since balancing request and acknowledgement pairs for an arbitrary number of clients requires an arbitrary number of transition events. In impure RTL we can take advantage of the values of the state variables **req** and **ack** to establish the balancing, as follows: $\forall i \exists j \enspace @val(req,i) = @val(ack,j) \wedge @(req,i) + d \geq @(ack,j)$

In an execution time environment, we'd like to actively check the satisfaction of the deadline, reporting an error if it is not met. The following ORE code for the client, an augmentation of the code exhibited above, serves this purpose:

```
proc client( id :  int )
<
   var reqtime :  time;
   event reqdeadline;
   when req = 0
      do { /* This is a critical section */
         req = id;
         reqtime = @( req, -1 ); /* reqtime is 'now' */
         oneshot( reqdeadline, reqtime + d ); /* a future event */
         };
   [ /* wait for the ack, watch for the deadline, in parallel */
       < /* one thread */
          when ack = id;
          .../* gather up the result */
          ack = 0; /* clear the acknowledge */
          break; /* terminate the sequence */
       >;
       < /* the parallel thread */
          watch reqdeadline
             do {
                var missed :  boolean = TRUE;
                    ix :  int = 1;
                while ix <= HISTORY && missed
                   {
                   if @val( ack, -ix ) = id &&
                       @( ack, -ix ) < reqtime + deadline
                      then missed = FALSE;
                   ix = ix + 1;
                   }
                if missed /* stop the other process and quit */
                   then { preempt; report-error( id ); }
                };
          break;
       >
   ]
>
```

The primitive **oneshot** used above sets a timer that will cause an event at a specific time in the future. This time is specified absolutely, not as a delay. Of course, if the time is already in the past, the event is triggered immediately. Notice that since we have associated the deadline detector with each client individually, we have an event associated with that specific task's acknowledgement deadline. This simplifies the deadline detection code because it doesn't have to worry about scanning for the appropriate deadline event, in addition to having to find a matching acknowledgement in the event history of the **ack** variable.

6.2 Sacred RTL - the specification ideal

The preceding subsection presented an approach for specifying timing properties of real-time systems by embedding RTL-like notation within ORE statements. The primary advantage of this approach is that it allows manipulation of the @ function and occurrence indices via ORE constructs. An alternative approach is to make the timing specification independent of the ORE language constructs. This can be viewed as superimposing RTL on top of ORE to annotate programs written in ORE with RTL formulae. Since the proposed system of annotation is independent of ORE syntax, a preprocessor can extract the timing information. One potential use of this information is to inform the scheduler of the timing requirements of a system. An alternative use is to detect a violation of timing properties at run-time. However, this approach requires separating the timing requirements of a system from its functional specifications. An RTL-based system of annotations is proposed in a report by Jahanian and Goyal[3]. A similar approach can be employed for annotating ORE with RTL formulae.

We introduce two syntactic language constructs for annotating ORE programs with RTL formulas: **assert** and **maintain**. The **assert** statement is used to test an RTL formula at a particular point in the execution of a specific ORE sequence. The **maintain** statement is used to enforce a constraint at all times in the execution of an ORE program. An assert statement is of the form:

```
    assert RTL-formula do S;
```

When an assert statement is reached in the execution of a sequence, the RTL formula is checked for satisfiability. If the formula in the assert is not satisfiable, the corresponding statement S is executed. Otherwise, the execution continues with the next statement. Observe that the satisfiability of the RTL formula is checked only when the assert statement is executed. A maintain statement is of the form:

```
    maintain RTL-formula do S;
```

The maintain statement ensures that RTL formula is tested whenever an event occurs that may affect the satisfiability of the formula. This is similar in a certain sense to the ORE statement

```
    when !RTL-formula do S;
```

except that the formula may contain quantifiers and may not contain state variables. The execution of the maintain statement can be viewed as starting a separate thread. The thread is blocked until the occurrence of an event which requires testing the RTL formula. The corresponding statement S is executed if a violation is detected.

Consider the requirement that a sensor must be sampled at least 10 times during each 500ms interval. This minimum sampling rate can be specified by the maintain statement in the following ORE program:

```
    proc sensor
        <
        .../* sample the sensor */
```

```
      <-SAMPLE
      .../* report sample result */
      >
   proc main;
   [
      sensor;
      <
      maintain $\forall i$ $@(SAMPLE, i+10) \le @(SAMPLE, i) + 500$ do
         v := ERROR;   /* report the error */
      >
   ]
```

6.3 Bounded History

Testing the satisfiability of an RTL formula in an assert or maintain statement requires recording of event occurrences in the past. Since checking a timing property may involve multiple occurrences of the same event, it may be necessary to keep more than one occurrence of an event at any point in the execution of a program. As described in earlier sections, the notion of a history in ORE allows remembering a finite history of occurrences of an event. However, the history size is defined as a constant at compile time. Furthermore, since an RTL formula may be arbitrarily quantified, it is impractical to examine all previous occurrences of an event when testing the satisfiability of a formula. For example, recall the RTL formula specifying a minimum separation of at least 100 time units between two consecutive occurrences of the event E: $\forall i\ @(E, i) + 100 \le @(E, i+1)$. The satisfiability of this formula can be tested at run-time if either the last two occurrences of event E are remembered or each occurrence of event E is kept for 100 time units.

For a discussion on the issues related to checking RTL formulas at run-time refer to the report[3]. In it, three classes of properties are identified that can be expressed in RTL and for which bounds can be established on event histories at compile-time. Algorithms for testing satisfiability of formulas in these classes are also described.

7 Concluding Remarks

This design exercise demonstrates that there is a good match between the facilities of ORE and the requirements of RTL for runtime implementation. In addition, the RTL-like formulae should provide an expressive tool for ORE programming. We plan to build this into ORE.

References

[1] Marc D. Donner and David H. Jameson. Language and operating system features for real-time programming. *Computing Systems*, 1(1):33–62, 1988.

[2] Marc D. Donner, David H. Jameson, and William Moran, Jr. Events: a structuring mechanism for a real-time runtime system. In *Proceedings of the Real-Time Systems Symposium*, 1989.

[3] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. In *Proc. of Fault-Tolerant Computing Symposium (FTCS-20)*, June 1990.

[4] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9), 1986.

Compiler Assisted Adaptive Scheduling in Real-Time Systems

Prabha Gopinath
psg@philabs.philips.com

Rajiv Gupta
gupta@philabs.philips.com

Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor
NY 10510

April 6, 1990

1 Introduction

In hard real-time applications, safety and timeliness of execution are critical issues since failures usually have unpleasant consequences. A task in such an application has constraints related to timing behavior, such as an Earliest-Start-Time (EST), a Deadline (DL), and a Worst Case Execution Time (WET)[1]. Tasks must execute subject to these time constraints and any failure to do so, a deadline failure for instance, may result in catastrophic results. This has perforce resulted in a conservative approach to scheduling such tasks; worst case execution times of tasks are assumed to be known *a priori* and a schedule is laid out on the basis of these WETs.

While such an approach, in general, ensures that tasks do meet their deadlines, it also results in severe under-utilization of the system since tasks typically complete in much less time (sometimes orders of magnitude less) than their WETs would indicate. This is an obvious drawback in an application with a very dynamic task set; tasks that could otherwise have been accommodated in a schedule are rejected. Another problem with the above naive approach to scheduling based on worst-case execution times arises when a task for some reason, perhaps owing to resource sharing delays, exceeds its WET. This results in deadline failure, but such a failure is noticed very late in the task's lifetime. This reduces the time available to take remedial action. One common solution to this is to add a safety margin by increasing the WET of a task by some arbitrary percentage in anticipation of resource sharing delays. This however only exacerbates the under-utilization problem while at the same time offering no assurance such margins would be adequate.

We propose a compiler assisted solution to this problem which, we expect, will mitigate some of the disadvantages of worst-case schedules. Through static analysis the compiler identifies and reorganizes the code of a task into partitions based on predictability and monotonicity, perhaps aided by programmer pragmas. It then inserts measurement code at selected partition boundaries. At run time, instantaneous task performance, as obtained

[1] A more complete taxonomy of time related notations can be found in the work by Haban and Shin [2].

by executing this measurement code and from measures of resource sharing delays, is used to modify the behavior of the remainder of the task and also to dynamically adapt the task schedules.

In Section 2 we describe the background of our research. In Section 3 we explain our approach and in Section 4 we compare it to other techniques. Finally in Section 5 we outline some future work including the use of statistical analysis to improve the run-time performance.

2 Background

Two projects have influenced this work. The first, by Haban and Shin [1,2], applies real-time monitoring of task execution, using dedicated hardware, to affect the scheduling of these tasks. The second, the Quartz Project at Univ. of Illinois, uses the notion of Imprecise Computations [4,6] - results of a poorer quality are tolerated - to ensure that hard real-time tasks are able to meet their deadlines. We explain both these in a little more detail.

In the first case, a real-time monitor consisting of dedicated hardware and software, is installed as a permanent part of a real-time system. A task in the system is programmer-divided into *n disjoint* partitions based on its structure. Code partitions may be classified as Predictable or UnPredictable depending on how accurately their WETs can be estimated statically. Resource sharing delays also introduce unpredictability. Programmer inserted events at partition boundaries are used by the monitor to compute actual execution time and hence the savings as compared to the WET. These savings are compared to the resource sharing delays and used to make scheduling decisions.

A monotonic algorithm is one where there is a monotonic relationship between the quality of the result generated and the duration for which the algorithm executes. Imprecise computation requires that a task encoding such a monotonic algorithm be divided into a mandatory subtask which must complete and one or more optional subtasks which may be aborted. Successful completion of the mandatory subtask yields an acceptable result of a poor quality. When the mandatory subtask completes, if there is sufficient time left in the schedule, the optional subtasks are allowed to continue. Each optional subtask progressively refines the result obtained from the mandatory subtask. They can also be viewed as *monotonically* improving upon the result generated by the mandatory, subtask.

3 Adaptive Scheduling

3.1 Introduction

In this section we discuss an approach to dealing with some of the problems with worst-case scheduling. We call this technique Compiler Assisted Adaptive Scheduling. Compiler techniques are used to partition and reorder application code based on criteria we present later. The compiler also inserts measurement code at selected locations. These locations are selected based on certain specific criteria that we present later. Delays and execution time savings are obtained at run time by time-stamping each time a task blocks at a resource and by executing compiler inserted measurement code fragments at appropriate points in the application code respectively.

3.2 Code Partitioning

Each task is viewed as a series of partitions that can be classified as being either:

- **Predictable**: These are computations whose execution time can be accurately determined statically. These include straight-line code blocks, conditionals where the different branches have approximately the same execution times, and loops with fixed, statically known bounds.
- **Unpredictable**: These are computations whose execution times cannot be accurately determined at compile time and include conditionals where the branches have significantly different execution times and loops where the loop bounds are not known statically.

The compiler is responsible for distinguishing predictable computations from unpredictable computations. Predictability of a code partition is determined by examining its execution time *variance* [7] which is computed through compile-time analysis. Code partitions with high variance are classified as being unpredictable.

In addition, and quite orthogonally, code partitions can also belong to one of the following classes:

- **Monotonic**: The code for a monotonic algorithm typically consists of an initialization part followed by a loop. The loop must be executed for some specified minimum number of iterations to obtain an acceptable solution. Further executions of the loop refine the solution. The execution time of the algorithm can therefore be deliberately varied by varying the number of iterations; reducing the number of iterations reduces the exection time; increasing the number of iterations increases the precision at the expense of execution time.
- **Non-Monotonic**: Straight-line code falls into this category. However we are primarily interested in loop behavior and classify a loop as being non-monotonic if it requires a certain number of iterations to obtain a result and further execution does not improve the result.

For each loop in a task the user can specify whether the loop represents a monotonic computation and also a desired minimum number and recommended number of iterations. This can be achieved by either extending the language or allowing the user to provide compile-time directives (pragmas). For example:

```
PRAGMA: Monotonic Loop MIN = 20; RECOMMENDED = 50
```

In practice it will not be possible to obtain a clean separation of a task into Predictable/Unpredictable or Monotonic/NonMonotonic partitions. There will be significant overlap between the two classifications as shown below:

1. **Unpredictable and Non-Monotonic (UP-NM)**: A code partition containing conditional branches whose alternate paths require varying execution times or non-monotonic loops with bounds unpredictable at compile-time are included in this category. The execution times of UP-NM partitions are unpredictable and cannot be varied at run-time.

2. **Unpredictable and Monotonic (UP-M)**: A monotonic loop whose body has unpredictable execution time. Each iteration of the loop is a UP-NM computation. Although the number of loop iterations of such a computation can be varied to control its execution time, the time required by a single loop iteration is unpredictable and cannot be varied at run-time.

3. **Predictable and NonMonotonic (P-NM)**: Computations whose execution time can be predicted accurately but cannot be varied at run-time fall in this category. Such computations contain straightline code, conditionals with alternate paths requiring approximately the same execution time, and loops with fixed bounds having loop bodies with predictable execution times.

4. **Predictable and Monotonic (P-M)**: Computations composed of loops whose bounds can be varied and whose bodies have predictable execution times belong to this category. Not only can the execution time of such a computation be varied at run time but it can be done fairly precisely as the loop bodies have predictable execution times.

3.3 Partition Reordering

The above list represents the classification scheme used by the compiler to divide a task into subcomputations. Next the compiler reorders the subcomputations [3] of a task according to the following principles.

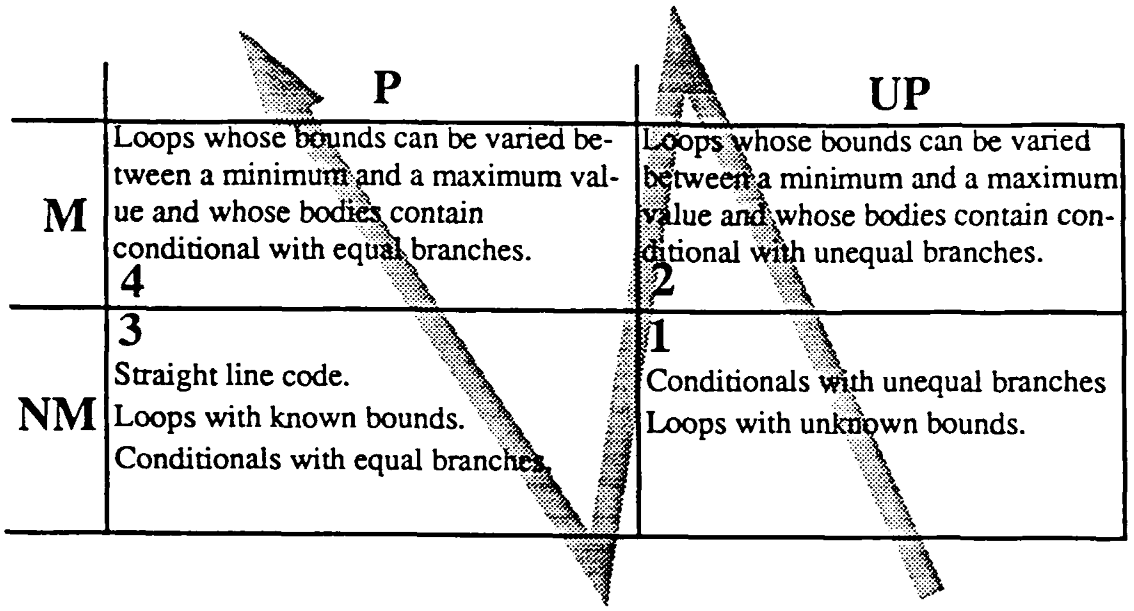
The unpredictable computations are performed before the predictable computations for two reasons. Firstly, if these computations execute in time much less than their WETs then the time savings can be used to adapt the schedules to accommodate additional tasks. Secondly, the accuracy with which the execution time of the remainder of the task can be predicted will be higher as it contains fewer unpredicatable computations. Since the execution time for the remainder of the task is what is critical in determining whether a task will meet its deadline, it is important for this information to be as accurate as possible.

Within the categories of predictable and unpredicatable computations the nonmonotonic computations should be executed before monotonic computations. This is because if nonmonotonic computations take longer to execute, the number of iterations of one or more monotonic loops can be lowered upto the MIN value specified in the associated pragmas in order to reduce the execution time of the remainder of the task. Figure 1 summarizes both the classification scheme and the sequence in which the compiler attempts to reorder the code.

3.4 Measurement

Once the compiler has identified and reordered code partitions based on the classification scheme shown above it inserts code fragments that, at run time, measure actual elapsed execution times and hence deviations from anticipated WETs. These code fragments update a data structure that contains attributes of all code partitions. The selection of measurement points determines the effectiveness with which the information collected can be used to ensure timely task completions.

We now discuss some criteria for selecting measurement points.



	P	UP
M	Loops whose bounds can be varied between a minimum and a maximum value and whose bodies contain conditional with equal branches. 4	Loops whose bounds can be varied between a minimum and a maximum value and whose bodies contain conditional with unequal branches. 2
NM	3 Straight line code. Loops with known bounds. Conditionals with equal branches.	1 Conditionals with unequal branches Loops with unknown bounds.

Figure 1: Code Classification and Reordering

- It is beneficial to perform measurements immediately after an unpredictable computation has been completed since this will enable us to determine if any time has been saved during its execution.
- Measurements should be performed prior to monotonic computations since if the task is behind schedule the number of iterations of a monotonic loop can be reduced to meet the deadline.
- The measurement points should be introduced less frequently at the beginning of the task and more frequently towards the end. This is because as execution proceeds the task gets closer to its deadline and is more likely to require adaptation to meet its deadline. This predicates greater precision in measurement.

There is a second class of measurement for which the compiler is not responsible and which is used to determine, at run time, the resource sharing and preemption delays experienced by an executing task. This is done by appropriately time-stamping the task each time it is blocked and unblocked.

3.5 An Example

To summarize, there are three main functions that the compiler performs. It identifies code partitions on the basis of the classification scheme mentioned earlier. It reorders the code partitions based on the criteria described earlier. Finally it selects partition boundaries that are most suitable for inserting measurement code.

The example in Figure 2 illustrates some of the aspects of the above process. The flowgraph has been partitioned into smaller computations and assigned to the code classification categories described earlier. The compiler has selected two points in the program at which to measure elapsed execution times. No measurement is performed after executing UP-NM(1) because it is too early in the task's execution to require adaptation. If the task

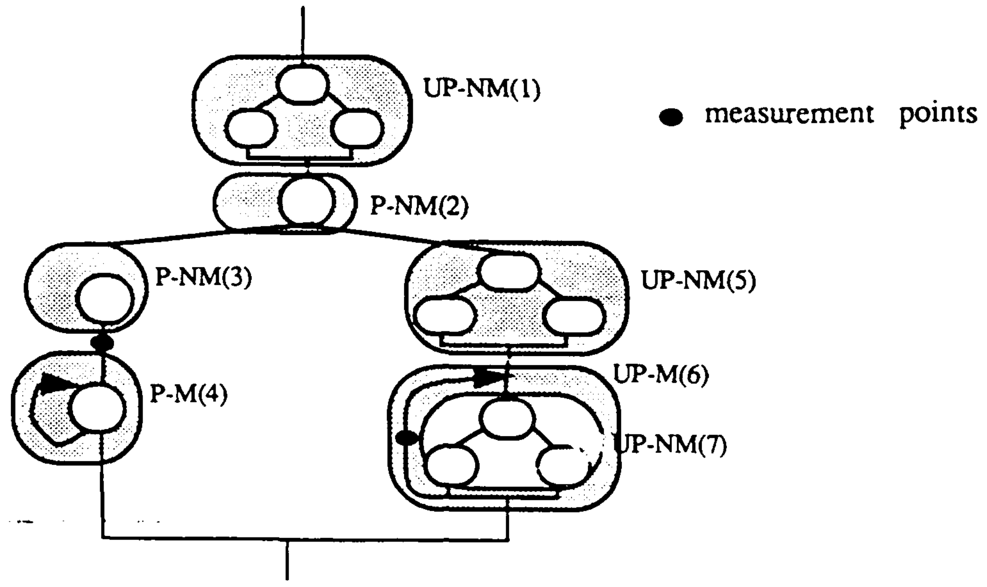


Figure 2: A Partitioned Control Flow Graph

follows the left branch the measurement is performed before P-M(4) because if the task is behind schedule the number of iterations of the monotonic loop P-M(4) can be reduced. Along the right branch another measurement point is introduced. The loop UP-M(6) can be viewed as a series of executions of UP-NM(7). Since each iteration of the loop body UP-NM(7) takes unpredictable amount of time to execute, the number of iterations of the loop cannot be accurately predicted prior to the execution of the loop. Therefore the measurement is performed after each iteration of the loop. This allows the maximum number of loop iterations to be executed without missing the task deadline.

3.6 Run-Time Adaptations

The previous sections discussed functionality related to the compiler, namely code partitioning, partition reordering, and insertion of measurement code. We now briefly describe how these phases interface with the execution phase of the task and how the information obtained can be used for adapting task behavior and task schedules.

The compiler stores attributes of all partitions that it has identified in a data structure called the Attribute Data Structure (ADS). This contains, among other things, WETs of all partitions, predictability and monotonicity characteristics, and MIN and RECOMMENDED values obtained from pragmas for all monotonic loops. The structure is organized to reflect the control flow graph of the task with paths weighted to reflect statistical correlations among various paths in the graph. The ADS resides at a fixed location in the data segment of the task and can be accessed by the task at run-time.

At run-time, each time the task executes one of the compiler inserted measurement fragments, the actual execution times of the relevant code partitions are obtained and inserted into the ADS. From these measurements the time savings, if any, can be computed. On the other hand each time a task is blocked at a resource, the delay that it experiences is measured. After each such resource-sharing delay the task checks whether the cumulative delay exceeds the cumulative savings. If so it looks ahead in the execution graph, by traversing the ADS, and checks all monotonic (UP-M, P-M) partitions to see whether changing one or more loop bounds will help to offset the cumulative delays. If not the task is aborted. If sufficient savings can be obtained from modifying monotonic loops than as many loops as are necessary are changed and the task continues. Again if sufficient savings are obtained the scheduler can decide whether to allow additional tasks into the schedule. By the same token if a task is aborted owing to excessive delays then a new task may be allowed in.

4 Comparisons

We now discuss some of the advantages of our work and compare it with the two projects mentioned earlier.

Unlike Haban and Shin's work our approach presumes no additional hardware support. While dedicated hardware such as they have used does improve the efficiency of monitoring, we feel that such an option is not readily accessible to the majority of installed applications. We expect that there will be some loss of efficiency since at run-time the task has to compute both time savings and resource sharing delays, both of which Haban and Shin do with the assistance of dedicated hardware. We address this problem by inserting measurement points very selectively and by designing the attribute data structure so that traversals are efficient. Again, by reordering the code so that unpredictable partitions are executed earlier, we gain time savings early in the task execution. This allows us to have fewer unnecessary aborts and the aborts that do occur will occur earlier in the task's execution thereby wasting less time.

As compared to the Imprecise Computation work of Project Quartz, we do not require that a monotonic algorithm be partitioned into one mandatory sub-task and one or more optional subtasks, but are able to use existing algorithms while still taking advantage of imprecision and monotonicity of algorithms. Since we are able to change loop bounds at run time, we can do so with greater precision than with the "chunking" inherent in partitioning into optional subtasks and with very little additional overhead. We hope to achieve most of the benefits of imprecision through the use of compile-time directives and compiler assistance rather than with a special-purpose programming language [5]. Thus our approach is more suited to revitalizing "dusty-deck" programs.

5 Future Work

We are currently working on several aspects of compiler assisted adaptive scheduling including an attribute grammar to better characterize task partitions, design of a data structure to maintain the attributes of the code partitions, and the use of statistical analysis to ensure high-performance run-time access to this structure. We now briefly touch upon the use of statistical information in adaptive scheduling.

At run-time, each time a task blocks and then is resumed it has to decide whether to abort or whether to continue execution after modifying the loop bounds of monotonic partitions that are further ahead in the execution sequence. At any point in the control flow graph (CFG) of a task there can potentially be any number of possible paths that the executing task can take. Each of these paths could contain monotonic partitions. In the absence of any other information the task would have to modify the loop bounds on all such monotonic partitions to obtain the necessary time savings. This however would be extremely inefficient, both in terms of traversal of the attribute data structure and in terms of actual modification of the loop bounds. One solution to this problem uses the compiler, again perhaps aided by pragmas, to determine correlations among various execution paths in the task cfg. Now when a task traverses the cfg, it modifies only those monotonic partitions that lie along paths that have a high correlation with the current execution point. In this way we hope to improve the dynamic behavior of run-time adaptations.

References

[1] Dieter Haban and Kang G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. In *Proceedings of the 10th Real-Time Systems Symposium*. IEEE, IEEE, 1989.

[2] Dieter Haban and Kang G. Shin. Monitoring Distributed Real-Time Systems and its Applications. In *Proceedings of the 6th Workshop on Real-Time Operating Systems*. IEEE, IEEE, 1989.

[3] D.J. Kuck, R.H.Kuhn, , D.A.Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimisations. In *Proceedings of 8th Symposium on Principles of Programming Languages*, pages 207–218. ACM, 1981.

[4] K.J. Lin, S. Natarajan, and J.W.S. Liu. Imprecise Results: Utilising Partial Computations in Real-Time Systems. In *Proceedings of the 7th Real-Time Systems Symposium*. IEEE, IEEE, 1987.

[5] Kwei-Jay Lin and Swaminathan Natarajan. Expressing and Maintaining Timing Constraints in Flex. In *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, pages 96–105. IEEE, 1988.

[6] J.W.S. Liu, K.J. Lin, C.L. Liu, and C.W. Gear. Research on Imprecise Computations in Project Quartz. In *Proceedings of the 1989 Workshop on Operating Systems for Mission Crtical Computing*. IDA, ONR, Univ. of Maryland, ONR, 1989.

[7] V. Sarkar. Determining Average Program Execution Times and Their Variance. In *Proceedings SIGPLAN'89 on Programming Language Design and Implementation*, pages 298–312. ACM., 1989.

Developing Software with Predictable Timing Behavior

Peter Puschner
Ralph Zainlinger

Institut für Technische Informatik
Technische Universität Wien
Treitlstr. 3/182
A-1040 Vienna, Austria

E-mail:
peter@vmars.UUCP
ralph@vmars.UUCP

April 2, 1990

Abstract

In critical real-time systems knowledge about the maximum execution times (MAXT) of tasks is of utmost importance. However, this knowledge is merely the first step towards building functioning systems that definitely meet their deadlines. Computing the MAXT of a task has to be an integral part of the application development and has to be directly incorporated into the implementation process.

This paper presents the concepts for an integrated programming environment that particularly supports the development of real-time software with predictable timing behavior.

We are grateful to Andreas Damm, Gustav Pospischil, Werner Schütz, and Alexander Vrchoticky for helpful discussions and comments on this report.

This work has been supported in part by the ESPRIT Basic Research Project 3092 "Predictably Dependable Computing Systems (PDCS)".

1 Introduction

Most computer systems for real-time process control must meet high standards of reliability, availability, and safety. In many of these real-time applications, the cost of a catastrophic system failure can far exceed the initial investment in the computer and the controlled object. To prevent such failures, system design must guarantee behavior as specified in the domains of both value *and* time during all anticipated operational situations.

One of the key aspects in designing predictable real-time systems is exact knowledge about the timing behavior of the real-time tasks involved. In order to guarantee that a real-time task can be finished before the expiration of its specified deadline, the maximum execution time (MAXT) or a reasonable upper bound have to be known in advance, i.e. before run time.

[Kli86, Pus89, Sha89, Sto87] present concepts on how MAXT can be calculated and which restrictions have to be imposed on programming languages for critical real-time systems (e.g. bounded loops).

This report suggests how these concepts could be effectively integrated into the design process of a real-time system. The major idea is that the MAXT analysis forms an integral part of the programming environment in order to provide the programmer with concise information about the maximum execution times of arbitrary pieces of code.

The proposed programming environment is conceived as part of an already existing prototype design environment for real-time system, the MARS Design System MARDS [Sen88, Sen89]. However, the concepts are not restricted to this particular design environment.

The structure of this report is as follows. Section 2 provides background information about the design principles applied in MARDS. Section 3 lists some basic requirements for the proposed programming environment, which is described in Section 4.

2 Background and Motivation

MARDS has been designed to support the development of critical real-time applications according to the MARS architecture [Kop89]. MARS (MAintainable Real-time System) is a periodic, time driven distributed real-time system architecture. A MARS system consists of a number of clusters each of which is built of a set of components interconnected by a real-time bus. Each component executes a set of tasks. Communication between tasks is realized exclusively through messages.

During design the overall system is gradually refined, first into a set of clusters (set of components characterized by a high inner connectivity), then each cluster independently of the others into components. Concurrently, the functional behavior of the system is modeled through *real-time transactions*. A real-time transaction refers to the execution of a set of correlated actions between a stimulus from the environment and the corresponding (time-constrained) response indicating the completion of this set of actions. During system design transactions are refined into subtransactions which finally become tasks.

It is a key concept of the design system and thus the underlying design methodology that time forms an integral part and does not constitute an isolated addendum. Transactions, for example, are characterized by various time attributes, e.g. MART, the maximum response time which is actually dictated by the environment, and MINT, the minimal interval between two invocations of the same transaction. At the task level MAXT determines a crucial characteristic. Before coding activities take place, MAXT is estimated for each task (by an experienced designer) in order to verify whether an appropriate pre run-time schedule can be found for the designed software or not. If the scheduling problem can be solved the tasks are coded respecting the estimated MAXTs.

3 Requirements for the Programming Environment

The following is a list of basic requirements for the proposed programming environment.

- **General Applicability:** Two fundamentally distinct approaches towards critical real-time task development are conceivable: One starting with a predetermined MAXT, which has to be met by the implementation (as a result of a consequent top down refinement), the other starting with the coding activities (bottom up) not respecting any given timing constraints during implementation. While the second strategy only demands calculating and adequately representing MAXT, the first approach requires more sophisticated mechanisms for comparing and controlling the desired and the actual MAXT as well as support for improving unsatisfactory results. The programming environment should support both approaches.

- **Timing Information:** During program development the programmer must have access to the following (time related) information at any point in time.

  - How long does the execution of this section of the program take ?
  - How long does the execution of the overall program take ?
  - Am I adhering to the demanded MAXT ?
  - If not, what is the time difference ?

- **Traceability:** Traceability in this context refers to the global consequences of changes in a particular (local) section of the program. Questions such as "What will happen to the global MAXT if I am able to reduce MAXT of this particular block" are likely to occur and have to be answered by the programming environment.
- **Modelability:** Modelability is closely related to traceability. Modelability in our definition comprises support for
  - comparing different combinations of implementation alternatives and
  - experimenting with hypothetical time changes in different program sections.
- **Turn Around Cycles:** The time it takes to compile and evaluate a program (in the time domain) has to be kept as short as possible. It is unacceptable to pass the code to a separate tool to calculate MAXT. All components (editor, compiler, MAXT analyzer) have to be integrated into a common framework.
- **User Interface:** A highly sophisticated (graphical) user-interface is required to appropriately handle and present the various timing information.

4 The Programming Environment

The requirements listed above are the basis for the programming environment. The programming environment consists of three cooperating parts, 1) an editor which incorporates a text editor and a so-called time editor, 2) a compiler and 3) a timing analysis tool. The components of the environment exchange timing information via a *timing tree* which represents the structure of the program or program fragment being developed. At each point in time the timing tree contains a certain amount of timing information. The amount of timing information present in the timing tree depends on which part of the environment has produced or changed the tree information.

Figure 1 shows the programming environment. The tools necessary for handling the timing information are presented in the grey area. The programmer implements his programs/tasks with the syntax directed text editor and passes them to the compiler.

The compiler generates the object code as well as the timing tree. The timing tree is a simplified program syntax tree augmented with timing information. Its nodes represent the sequential parts, branches, bounded loops, markers and scopes (the latter have been introduced in [Pus89]) of programs.

The timing information produced by the compiler contains only the maximum execution times of sequential (basic) blocks, i.e. the compiler fills the leaves of the tree with timing information. The timing tree is forwarded to the MAXT analyzer which fills the inner nodes with the values calculated for the maximum execution time of the respective

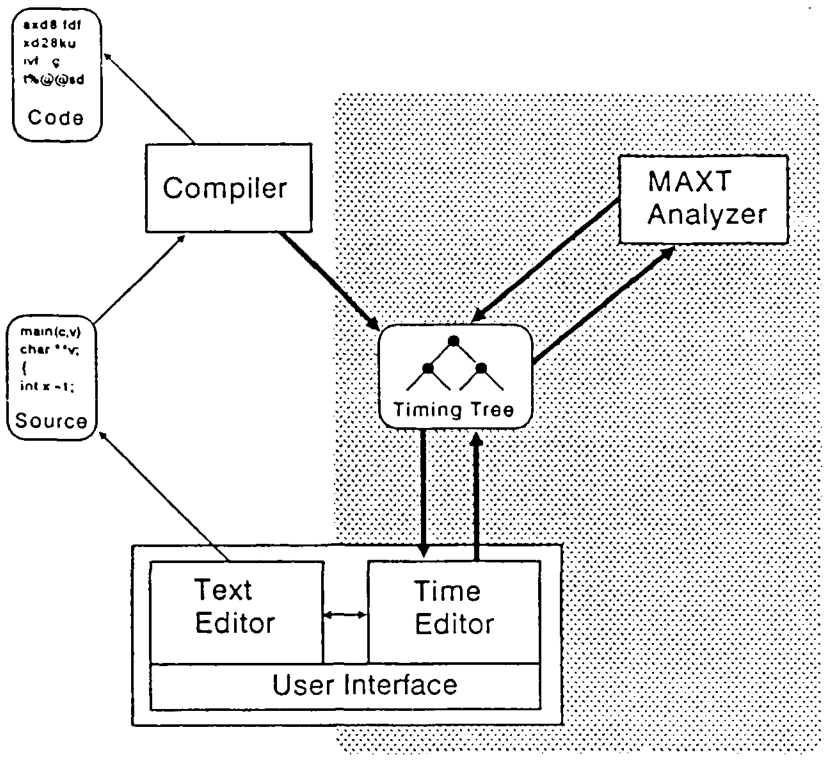


Figure 1: programming environment for time guaranteed tasks

program parts represented by this subtree. Thus, the maximum execution times for all branches, loops, scopes and the whole piece of source code analyzed are available in the timing tree.

The *time editor* reads the timing tree after it has been completed by the MAXT analyzer. The time editor transforms the time information contained in this tree into a Nassi-Shneiderman diagram which is displayed in parallel with the program code. Every structural unit of the diagram provides a display field for the actual MAXT of the matching program part and one or more display fields for hypothetic timing values.

Time Editing

If the user is not satisfied with the timing behavior of his program he can set hypothetic execution times at every level of the Nassi-Shneiderman diagram and have the MAXT calculated under the assumption of the hypothetic values. In this way he can model the timing behavior of potential improvements and gets an immediate feedback about its gains before actually changing any piece of code.

If a simulated value is input at a specific level in the Nassi-Shneiderman diagram both actual and hypothetic time values of the abstraction levels below are ignored in the MAXT calculation. A change at one specific level influences all simulated timing values of the surrounding constructs. Hypothetic timing values which have been set by the programmer are marked to inform the programmer that these values are not derived from analyzing actual program statements.

Code Variants

Often the programmer has many possibilities for writing tasks with the same functionality. It might not always be obvious which variant needs the minimal amount of time to execute. For this reason the environment supports multiple windows in which different versions of program parts can be edited and evaluated. The programmer may code different variants of some program parts or even implement different algorithms, calculate their maximum execution time and check which variant is suited best. The least time consuming one may then be used in the application being developed.

5 Summary and Conclusion

In this report we have discussed the requirements and concepts for a real-time task programming environment. When developing software with predictable timing behavior it is a waste of efforts to write programs and *find out afterwards that they violate the* specified timing behavior. Therefore, the timing behavior has to be a central issue throughout the whole task development process.

We have proposed a way to include timing behavior in the task design. Timing information is available during the whole implementation process. In every phase, the task's timing behavior can be calculated and modeled, thus allowing the programmer to test whether the specified requirements can be met. In addition, the programming environment allows the user to generate different variants for a piece of source code to find the implementation with the shortest MAXT.

References

[Kli86] E. Kligerman and A. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, Sep. 1986.

[Kop89] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[Pus89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[Sen88] Ch. Senft. A Computer-Aided Design Environment for Distributed Realtime Systems. In *IEEE Compeuro 88, System Design: Concepts, Methods and Tools*, pages 288–297, Brussels, Belgium, Apr. 1988.

[Sen89] Ch. Senft and R. Zainlinger. A Graphical Design Environment for Distributed Real-Time Systems. In *Proc. 22nd Annual Hawaii International Conference on System Sciences, Vol. II*, pages 871–880, Hawaii, Jan. 1989.

[Sha89] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[Sto87] A. Stoyenko. *A Real-Time Language With A Schedulability Analyzer*. PhD Thesis, Computer Systems Research Institute, University of Toronto, Dec. 1987. Technical Report CSRI-206.

A Synthetic Workload for Real-time Systems

Daniel L. Kiskis and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122

ABSTRACT

This paper presents our progress in the development of a synthetic workload (SW) for real-time systems. We discuss the issues involved in developing a SW and identify a number of important properties a SW should possess. Then, we outline the approach we have taken to design and implement a SW with these properties. Also, described briefly is the workload model upon which the SW is based. Finally, we discuss the preliminary version of the SW which has been constructed for HARTS, an experimental distributed real-time system being built at the Real-Time Computing Laboratory, The University of Michigan.

1 Introduction

For critical real-time applications, it is essential that a computer system meet its performance and dependability requirements. The ability of the system to meet these requirements can be partially determined *a priori* through modeling and simulation. However, simulation is often inadequate because simulation models lack sufficient detail about the system. For tractability purposes, many system details are abstracted out in the simulation model. Hence, for measurements to be accurate, they must be made on the actual system.

An example of where these effects are important is in the study of the schedulability of tasks on a real-time system where both periodic and aperiodic tasks are present. Some results for specific cases may be arrived at analytically or through simulation. However, most analytical solutions and simulation approaches assume independent tasks. They ignore resource contention between tasks. This assumption is rarely valid in real systems. Simulation study can partially account for the effects of resource contention delays, but only through experimental evaluation on the target system may the total effect of resource contention be seen.

Given the importance of experimental evaluation, we must identify factors which particularly influence performance and dependability. One of the most important factors is workload. The *workload* is the set of programs that the real-time system is executing along with the input data associated with those programs. The demands for resources placed on the system by the workload

The work reported in this report was supported in part by the NASA under Grant No. NAG-1-296 and NAG-1-492 and the Office of Naval Research under Contract No. N00014-85-K-0122. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the NASA or the ONR.

are called the *workload characteristics*. For the analysis of the system to be valid, the workload which the system is executing must be taken into consideration. It is known that performance and dependability are strongly dependent on workload. For example, Woodbury and Shin [1] showed the relationship between fault latency and workload. Fault latency is the time between the occurrence of a fault and the subsequent generation of an error by that fault. In a fault-tolerant computer, fault latency affects dependability. Longer fault latency increases the probability that there will be multiple latent faults in the system at one time, a situation which may result in system failure. They demonstrated that fault latency, and thus dependability, is a function of system utilization, a workload characteristic. As utilization increases, fault latency decreases, and, thus, dependability improves.

There is a great need for tools which will allow the user to perform experiments on a system with a known workload. In particular, the tools should support user specification of workload characteristics along with measurement of selected performance and dependability indices. In this way, the relationship between specific workload characteristics and system performance and dependability may be observed. A number of tools, such as job mixes, kernels, workload models, and benchmarks, have been developed which act as workloads for the system under study and/or perform measurements of a computer system. Each of these tools is useful for providing measurements at a given level of detail.

The problem with these tools is that they are often not very flexible. They cannot easily be altered to exhibit a range of workload characteristics. Another problem with them is that, with few exceptions, they were each developed for use on non-real-time systems. As such, they often incapable of being representative of real-time workloads. A real-time workload has a number of distinguishing properties, such as periodic tasks and hard deadlines, which are not present in non-real-time systems. We propose to remedy the shortcomings of these tools through the development of a tool which will provide greater flexibility and greater representativeness for real-time systems. The tool to be used is a *synthetic workload* (SW). A SW consists of a set of programs which are written such that their structure and behavior models that of the programs in a real-time application. The tasks of the SW are parameterized such that the workload characteristics may be easily changed to suit the experimenter's and applications' needs.

This paper identifies the desired properties of a SW and describes our efforts to develop one SW with these properties for use on distributed real-time systems. In the past, little effort has been made to establish a structured, scientific basis for the design and implementation of SW's; that is, most of the work has been *ad hoc*. To remedy this deficiency, we propose a structured approach to SW design based on a real-time workload model.

The workload model will provide a high-level specification of the structure and behavior of the SW. It will also describe the system workload in sufficient detail to be used in analytical evaluation. In this paper we will concentrate on the model's use as a specification of the SW. To implement the SW, we must also define the low-level details. Hence, the implementation of the SW will require two steps. The first step is the specification of the software structure of the SW. It entails specifying what processes must execute on each processor and defining the communication and synchronization between the processes on the same or on separate processors. The second step will be the development of a high-level language for specifying the SW for the entire system. Such a language will increase the utility of the SW by making it feasible for the user to specify a SW for a large distributed system. A specification in this language will provide the structure and parameters of the synthetic tasks along with their interactions. It will be compiled by a synthetic workload generator (SWG) into the individual tasks and the necessary control processes for each processor.

2 The Synthetic Workload

A synthetic workload should possess a number of properties. The most important of these are representativeness, flexibility, simplicity of construction, compactness, and system independence. Definitions of these may be found in [2]. Probably the most important of these properties is representativeness. Many performance evaluations are aimed at determining how the system will behave once it is put into operation. As such, accurate measurements can only be obtained if the SW that is executed is representative of the actual workload the system will execute. To improve the representativeness of the SW, the workload model will be described using a dataflow notation. A dataflow notation has been chosen so as to be compatible with the notations used in rapid prototyping and high-level specification of real-time systems. As such, the SW will be structurally similar to the actual workload. This compatibility has been shown to produce representative SW's [3]. Since many of the functions of the workload will be abstracted out, the SW will be more compact than an application based on the same notation.

We have completed the first step in the implementation of the SW. The software structure has been designed and implemented based on the workload model and the requirements of the target system. The target system for the SW is the Hexagonal Architecture for Real-Time Systems (HARTS), a distributed real-time multiprocessor system being developed at the Real-Time Computing Laboratory (RTCL) at the University of Michigan. An architectural description of HARTS can be found in [4] and its operating system, called HARTOS, is described in [5]. The structure of the SW is based loosely on the SW developed for HARTS by Woodbury [6].

The SW implements the workload model as a collection of processes on each processor. This collection consists of two groups: the *driver* processes and the *application* tasks[1]. The application tasks are the main components of the SW and are responsible for producing the resource demands on the system. The driver processes are mostly responsible for initializing and starting the workload and for the collection of performance data.

The four processes in the driver are the *root* process, the *logdata* process, the *dispatcher* process, and the *trigger* process. The root process spawns all the other processes and tasks in the system and establishes the communication mechanisms through which the processes may communicate. The logdata process collects performance data and stores it in memory. This data may later be read for off-line analysis. The dispatcher is used to model the occurrence of external events which effect the behavior of the workload. For example, the dispatcher may simulate the arrival of sensor data. It is also used to provide task control which is not supported by the target system, but which is required for a given experiment. For example, on HARTS, the dispatcher is used to provide periodic task invocation because the current HARTOS does not yet support periodic task scheduling. The trigger acts as a software timer for the dispatcher.

The structure of the driver is not influenced by the values of the workload parameters. It

[1]The term "task" is used here to distinguish the application processes from the driver processes.

is constant regardless of the experiment which is being performed. The characteristics of the workload are changed by altering the structure and behavior parameters of the tasks. Tasks have a basic skeleton structure which is required for interaction with the dispatcher. Otherwise, the structure of the task is completely flexible. The task parameters specify the scheduling characteristics (e.g., period, deadline, etc.) and behavior characteristics (e.g., resource usage) of the tasks. By parameterizing the task structure, we improve the flexibility of the SW. Ease of construction is improved by locating the parameters in a common table where they may be readily altered by the user.

3 The Synthetic Workload Generator

The SW is being designed to be used in a distributed real-time system. In such a system, each processor must execute both the driver processes and application tasks. The driver processes are identical on each processor. However, in all but the most trivial cases, the application tasks need to be different for each processor. These differences reflect the heterogeneous nature of distributed real-time systems. The tasks will differ depending on which resources are available on a particular processor. They will also differ to model varying load patterns in the system. Even in the case of tasks which are structurally identical, there will be processor-dependent parameters which will differ between the tasks.

In the current version of the SW, application tasks must be individually coded in C and debugged. This process is tedious and error-prone. For a system as large as the current version of HARTS with 19 nodes, this process becomes especially difficult. It would require creating and debugging the application tasks for up to 57 different processors. Such a design goes against the second and third properties of a good SW, i.e., flexibility and simplicity of construction. What is needed is a high-level Synthetic Workload Specification Language (SWSL) to describe the workload in terms of the workload model. The SWSL description will be compiled by a SWG which will produce the parameter tables and the C code for the individual application tasks.

The SWSL will be a representation of the dataflow notation in a textual form. Its purpose is to define a workload in terms of synthetic tasks and data flows. Since most, if not all, of the computation and logic are abstracted out of the tasks, the SWSL need not possess the complexity and expressive power of a true programming !anguage. The only constructs required are those which specify the components of the workload model. The language will allow the specification of implementation details such as task to processor assignment and resource assignments.

A number of additional features will be included in the language. The SWG will have access to a library of common operations. This library will consist of generic and user-definable operations which may be used within a synthetic task specification. The language will also allow the user to define multiple instances of a synthetic task. The user will write the definition of the generic instance of the task and specify to which processor each instance is to be assigned. The SWG wil' use this information to generate the code for each instance of the task.

One benefit of the SWSL is that it will provide a system independent specification of the workload. It will assume no specific target system. As such, for purposes of comparison, the same workload specification can be used to generate SW's for different systems.

4 Summary and Conclusions

We have identified a number of properties that a SW must possess. These properties are important because they improve the usefulness and the ease of use of the SW. Our approach to designing and implementing a SW is aimed at incorporating these properties. The SW is based on a workload model which allows system independence and analytical verifiability. To create a SW for a particular study, the workload model will be described using the SWSL. The SWSL improves the ease of construction of the SW by allowing the user to define the SW's structure at a high level. This description will be compiled by the SWG to create the SW code. This code is then executed on the system and the appropriate measurements made.

Although our SWG is targeted to create SW code for HARTS, the SWSL is system independent and the basic structure of the SW processes will be portable to other systems. Our implementation enables us to determine what engineering decisions must be made to implement the SW on any realistic distributed real-time system.

References

[1] M. H. Woodbury and K. G. Shin, "Measurement and analysis of workload effects on fault latency in real-time systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 212–216, February 1990.

[2] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, 1978.

[3] M. Calzarossa, M. Italiani, and G. Serazzi, "A workload model representative of static and dynamic characteristics," *Acta Informatica*, vol. 23, no. 3, pp. 255–266, June 1986.

[4] M. S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. on Comput.*, vol. C–39, no. 1, pp. 10–18, January 1990.

[5] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: A distributed real-time operating system," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 72–89, July 1989.

[6] M. H. Woodbury, *Workload Characterization of Real-Time Computing Systems*, PhD thesis, University of Michigan, 1988.

Real-Time Mentat Programming Language and Architecture†

Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

Ami Silberman and Jane W. S. Liu
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

I. Introduction

By a *real-time (computing) system*, we mean one in which the execution of some or all of its programs must satisfy timing constraints. The next generation real-time systems are likely to be built on parallel and distributed architectures, ranging from multiprocessors and multicomputer systems to loose-coupled host computers and workstations. A problem with using many of these systems is that the programmer must explicitly manage the different aspects of the parallel environment: message passing, synchronization, and other details. Unless the programmer is partially relieved of this burden, it is difficult to program parallel (and distributed) real-time applications.

Real-Time Mentat (RTM) is a programming environment designed to simplify the task of programming parallel real-time applications. Specifically, it is an object-oriented and data-driven system whose architecture is as shown in Figure 1. It provides an easy-to-use, transparent mechanism to exploit parallelism and run-time support for scheduling and executing parallel, real-time programs. The RTM programming language is an extended C++. The extensions are added to facilitate automatic detection of data flow and generation of data-flow graphs, to express the timing constraints of individual granules of computation, and to provide scheduling directives for the run-time system. In addition, through its visual user interface, which allows the programmer to visualize and modify any program graph and, hence, the corresponding RTM program, Real-Time Mentat *supports the visual programming paradigm*.

This paper discusses the design choices made in Real-Time Mentat and gives a high-level view of its architecture and programming language. The rest of this paper is divided into four sections. Section II discusses the design objectives of Real-Time Mentat and the approaches taken to meeting these objectives. Section III describes the underlying computation model and the object-oriented approach to

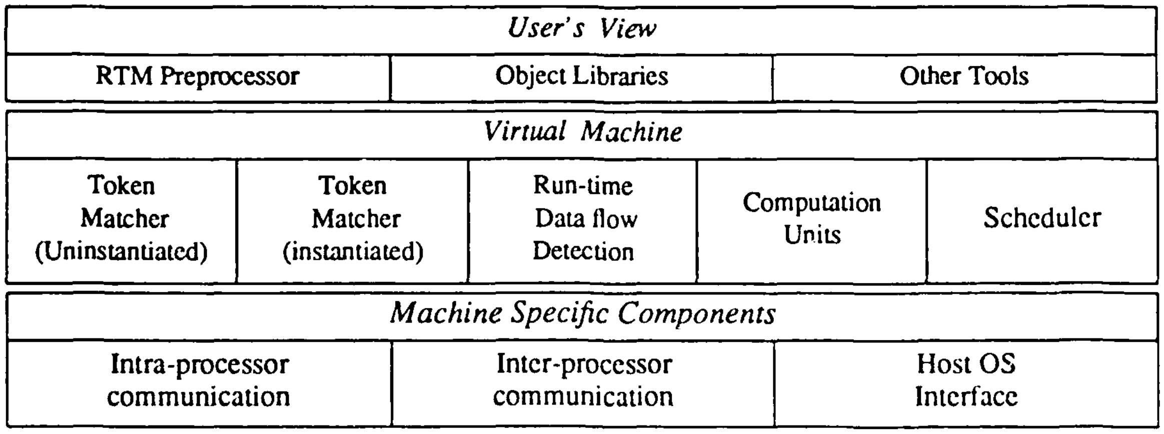

User's View
RTM Preprocessor
Object Libraries
Other Tools
Virtual Machine
Token Matcher (Uninstantiated)
Token Matcher (instantiated)
Run-time Data flow Detection
Computation Units
Scheduler
Machine Specific Components
Intra-processor communication
Inter-processor communication
Host OS Interface

Figure 1. Real-Time Mentat Architecture

† This work was partially supported by the NASA Contract No. NAG 1 613 and the U.S. Navy ONR Contract No. N00014-89-J-1181.

its implementation. Section IV describes the features of the RTM programming language. Section V briefly describes the RTM Virtual Machine and summarizes its status.

II. Design Objectives and Choices

One of the primary design objectives of Real-Time Mentat is that it would relieve the programmer of the burden of explicitly managing parallel computations. It takes a hybrid approach to meet this objective: the programmer explicitly specifies those object classes that have sufficiently large granularities and, thus, defines the granules of computation that can potentially be executed efficiently in parallel on different processors. However, the data dependencies and parallel structures of the computation are implicit. Invocation, communication, and synchronization are handled automatically by the compiler and the run-time system. The compiler constructs some graphs at compile time and generates code to automatically construct other data-flow graphs at run time.

The other design objective is that it would provide a flexible interface between parallel, real-time applications and the underlying system. Real-Time Mentat again takes a hybrid approach. Specifically, Real-Time Mentat permits the expression of the timing requirements of its programs in terms of timing constraints similar to those used in [1,2]. The programmer tells the underlying system the timing constraints of the individual granules of computation, but the system has the burden of scheduling the granules to meet these constraints. Timing constraints can be either *soft* or *hard*. Rather than eliminating non-determinism in timing requirements completely (as, for example, in Real-Time Euclid [3]), only the granules of computation with hard timing constraints are restricted to have bounded and predictable execution time and resource requirements that can be determined at compile time. Using the timing constraints as scheduling directives, the RTM run-time system guarantees that all hard timing constraints are met while trying to meet as many soft timing constraints as possible. Moreover, RTM programming language makes it relatively easy to implement imprecise computations [4,5] and primary and alternative versions [6] that have different execution times and, thus, provides the run-time system with flexibility in scheduling and resource management.

III. The Real-Time, Macro-Data-Flow Model and Its Implementation

Real-Time Mentat is based the macro data-flow model of computation [7-9]. Individual granules of computation that are treated by the underlying system as units of work to be assigned to processor and scheduled for execution are *macro actors*, hereafter simply referred to as actors. Actors perform high-level functions, such as FFT, encrypt, and database_read; their granularity is chosen by the programmer using a language construct described in Section IV. Similar to traditional large-grain data-flow models [10,11], computation is data driven. A high-level view of each program is its macro data-flow graph, hereafter referred to as its program graph. Nodes in this graph are actors. There is a directed arc from one actor to another in the graph when there is data dependency between them. Tokens, containing data and control information, are sent along the arcs. When matching tokens on all its input arcs have arrived, an actor is ready for execution. When an actor executes, it consumes the input tokens. When its execution completes, it generates one or more output tokens containing its result. The output tokens are forwarded along the arcs from it to the actors that depend on the result.

The macro data-flow model differs from the traditional large-grain data-flow models in two ways. First, in addition to *regular* actors which do not maintain state information between executions, some actors, called *persistent actors*, do so. Persistent actors provide the ability lacking in the traditional models to model side effects and permit inter-program communication. This capability is necessary for applications such as data communication and database management. Persistent actors are similar to resource managers in [12], but differ from resource managers in that several actors may share the same persistent state. This corresponds to an object class with several member functions in its interface. Second, while program graphs in tradition models are either entirely static or dynamic only to the extent of allowing variable number of copies of some subgraphs, the structures of macro data-flow graphs can change dynamically as graph nodes (actors) are elaborated at run time into arbitrary subgraphs.

Specifically, program graphs are represented using data structures called *futures* and *future lists* [7-9]. A future list is a list of futures. Each actor receives a future list with its input tokens; each future in the list names a *dependent actor* that will receive a copy of the result produced by the actor. The list sent to a dependent actor describes a directed graph rooted at that actor. Actors may augment the future lists that are passed to them. When the execution of an actor terminates, it sends a new future list, together with its result, to each of the dependent actors named in its future list. In this way, the node representing an actor in the program graph can be hierarchically expanded. We call this subgraph the *elaborated subgraph* of the actor. The modification of the program graph via elaboration is a local operation, and other portions of the graph are not affected.

Similar to other real-time graph models (e.g., [13,14]), each Mentat actor may have its own timing constraints. In order for the system to be able to guarantee that hard timing constraints are met, graph elaboration must be restricted for actors with this type of constraint. Specifically, every actor (and subgraph) with a hard timing constraint must have bounded execution time and resource requirements that are determined from static analysis at compile time. Moreover, the set of actors with hard timing constraints and the portions of the all program graphs containing them are known to the run-time system before their execution begins. This makes it possible for the scheduler to determine the schedulability of all these actors and the strategy used to schedule them. An actor (or subgraph) with a soft timing constraint, on the other hand, can be elaborated dynamically at run time into different subgraphs. When the imprecise computation technique [4,5] is used, each actor with a hard timing constraint is expanded into two parts: its mandatory part and its optional part; they have hard and soft timing constraints, respectively.

In Real-Time Mentat, actors are realized as external operations of RTM objects. Each RTM object has a name, a representation of the data stored in the object, a set of externally visible operations, a process executing in parallel with invocations of operations of the object, and a (optional) set of timing constraints. Each actor is implemented by an operation of some object, or, in the case of an actor with multiple versions, by operations of several different objects. To invoke an operation of an object, tokens (messages) are sent to the object, one for each parameter. When the tokens for all parameters have arrived for a particular operation, the corresponding actor is enabled, and the execution of the operation may begin. Each token contains a computation tag that is used to match the token with other tokens belonging to the same sub-computation. A computation tag contains the system-wide unique name for the specific object instance to which is the token is sent, a computation (sequence) number that specifies to which sub-computation of the named object this token belongs, and the timing constraints that give the intervals of time within which the operation is constrained to execute. The computation number in a computation tag is a unique label of a node in an elaborated program subgraph. Computation tags are similar to, and provide the same function as, token colorings in traditional data flow systems [15].

IV. Real-Time Mentat Programming Language

The RTM language is obtained by adding six extensions to the C++ programming language. They are (1) the keywords **Mentat** and **persistent** in class descriptions, (2) the member function **main()** in class definitions, and (3) **creat()/destroy()** statements, (4) return to future (**rtf()**), (5) **select/accept** statements, (6) *timing blocks* and timing constraints. The RTM preprocessor translates RTM programs into C++ programs extended with library calls.

To provide the programmer with a way to control the degree of parallelism, Real-Time Mentat allows both standard C++ classes and Mentat classes to be defined. By default, a standard C++ class definition defines a standard C++ class. The programmer defines a Mentat class by using the keyword **Mentat** in the class definition. This defines a set of actors, one for each external operation, that can potentially be executed in parallel with other actors in order to improve performance. Operations that are not complex enough to yield a sufficiently high ratio of the execution time to average message passing time cannot be efficiently executed in parallel due to high communication overhead. Classes of objects with such external operations should be defined as standard C++ classes. A Mentat class may be declared as either **persistent or regular**. The programmer may specify a member function **main()** in the class definition of a persistent class. The main() procedure is started by the underlying machine once the object has been instantiated. It represents the thread of control in the object, and when it terminates the object is destroyed.

Transparent to the user, a Mentat class definition is really two class definitions. One defines a *Mentat variable*, the interface of the Mentat class that the user of the class sees. The other defines a class, the instances of which implement the Mentat class. We call these instances *Mentat objects*. This feature, together with the **creat()/destroy()** functions described below, makes it possible to define *generic* instances, that is, instances of Mentat classes represented by their names rather than their physical selves. When a new Mentat variable is declared, a new instance of the object class is not automatically instantiated. Instead, only an unbounded object name of the appropriate type is instantiated. To provide the programmers with a means to create new instances of Mentat objects, we have added two new reserved member functions for all Mentat class object: **create()** and **destroy()**. These functions are used to instantiate new instances and destroy existing instances of Mentat classes. In particular, **create()** allows the programmer to provide optional location hints (e.g., high_communication_ratio, co_locate, etc.) that inform the underlying system where, ideally, the new instance is to be instantiated, e.g., on a different processor or on the same processor as some other Mentat objects. The location hints allows the programmer to influence the underlying system's decision on where the new object is actually instantiated.

The function **rtf()** is the RTM analog to the **return()** of C++. Its purpose is to allow each RTM member function (actor) to return a value to its successor nodes in the program graph. It takes two types of arguments, local variables or constants and subgraphs. Returning to a subgraph using **rtf()** is the mechanism for subgraph elaboration. Their use is transparent to the programmer except for the case where the actor has hard timing constraints.

In standard C++, member functions of an object are always available. RTM objects must be able to specify which operations are candidates for firing. **Select/accept** statements are added for this purpose. Specifically, these statements allow the programmer to specify those member functions that are candidates for execution based upon a broad range of criteria, including timing constraints of the computation. The semantics of RTM select statement is similar to the semantics of select statement of ADA. A RTM accept statement may list two alternative fct-declarators. The effect is that a call is made to either one of the declared functions. Which function is called is decided at run time by the scheduler depending on the current timing constraints of the call. If one of the options can be completed in time while the other cannot, the one that can be is chosen. If both can be completed in time, the first one is chosen. In this way, the programmer can specify explicitly the different operations, with different timing requirements, that implement the same actor.

The syntax of a RTM timing block is same as in Flex [2]:

```
        timing_constraint ~> exception action
            begin <statements> end;
```

A timing constraint is a Boolean expression containing one or more of the following keywords: **start, deadline, duration**, and **period**. (They are, respectively, the time before which execution of the block must begin, the time at which its execution must end, the length of time interval during which the block may execute, the length of intervals during each of which the block must execute once.) A timing constraint is considered hard unless it is declared to be otherwise by the keyword **soft**. When the expression evaluates as true, the exception action associated with it is executed. A legal exception is **rtf()**. This makes it possible to return an immediate result produced so far in the course of the computation if the execution of the timing block must be prematurely terminated.

V. RTM Virtual Machine and Current Status

To support parallel execution of RTM programs on a multiprocessor or distributed system, a RTM virtual machine runs on each processor. The RTM virtual machine is based on the existing Mentat virtual machine, which has been implemented to run on the Encore Multimax as well as hypercubes and networked Sun workstations [7-9]. Conceptually, the structure of the RTM virtual machine is similar to that of a traditional data-flow machine; it contains storage units for objects, tokens and predicates; a matching unit; a computation unit; and an update units. The predicate storage unit stores a current predicate and a message queue for each object in the object storage unit. The predicate specifies which actors of the object are available for execution. The token storage unit is responsible for ensuring that tokens are delivered to the correct actors. The update unit is responsible for forwarding the results of the computation to the actors named in the future list and updating the predicate storage unit with a new predicate. When a new message arrives for an actor, the matching unit determines if that message enables the actor by applying the object's predicate to the messages stored in the token storage unit. Once the matching unit has determined that an actor is enabled, it places that actor in the ready-to-run queue in the computation unit. The computation unit is responsible for executing actors that have been enabled and placed on the ready-to-run queue. It examines the ready-to-run queue and selects an actor to execute based on the timing constraints of the actors in the queue. It then executes the actor until the actor completes (or until its timing constraint is violated). Once the execution of an actor is terminated, the computation unit notifies the update unit and goes to examine the ready-to-run queue again.

The RTM virtual machine differs from the existing one in the ways tokens are sent, ready-to-run queue is ordered, and the matching and computation units operate. In Real-Time Mentat, the update unit may preferentially forward and update the results from computations according to the underlying scheduling policy. Similarly, the ready-to-run queue is ordered according to the scheduling policy adopted by the system. The matching unit keeps tracks of the timing constraints of actors who have not yet received all of their input tokens and ensures the appropriate exception handling if any of these constraints is violated.

A prototype RTM system is being implemented on a 10-processor Encore Multimax and on a network of Sun workstations. A visual interface is being added to support visual programming of RTM programs. Once the prototype system is completed, we plan to collect and develop a set of real-time applications and use them in the evaluation of the functionality and performance of this prototype system.

References

[1] Lee, I. and V. Gehlot, "Language Constructs for Distributed Real-Time Programming," *Proceedings of Real-Time Systems Symposium*, December 1985.

[2] Lin, K. J. and S. Natarajan, "Expressing and Maintaining Timing Constraints in FLEX," *Proceedings of Real-Time Systems Symposium*, December 1988.

[3] Klingerman, E. and A. D. Stoyenko, "Real-time Euclid: A Language for Reliable Real-time Systems," *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[4] Lin, K. J., J. W. S. Liu, and S. Natarajan, "Concord: A System of Imprecise Computations," *Proceedings of the 1987 IEEE Compsac*, pp. 75-81, Tokyo, Japan, October 7-9, 1987.

[5] J. Y. Chung, J. W. S. Liu, and K. J. Lin, "Algorithms for Scheduling Periodic Jobs That Allows Imprecise Results," *IEEE Transactions on Computers*, to appear

[6] R. H. Campbell.

[7] Grimshaw, A. S. and J. W. S. Liu, "Mentat, An Object-Oriented Macro Data Flow System," *Proceedings of the ACM 2rd Conference on Object-Oriented Programming Systems, Languages, and Architecture* pp. 35-47, Orlando, Florida, October 6-8, 1987.

[8] Grimshaw, A. S. and J. W. S. Liu, "The Mentat Programming Language and Architecture," *Proceedings of Workshop on Future Trends of Distributed Computing Systems in the 1990's*, pp. 426-437, Hong Kong, September 14-16, 1988.

[9] Grimshaw, A. S., "Mentat: An Object-Oriented Macro Dataflow System," Ph.D. thesis, Technical Report No. UIUCDCS-R-88-1440, Department of Computer Science, University of Illinois, June 1988.

[10] Brock, J. D., Omondi, A. R., and D. A. Plaisted, "A Multiprocessor Architecture for Medium-Grain Parallelism," *Proceedings of the 6th International Conference on Distributed Systems*, pp.167-174, May 1986.

[11] Babb, R. F., "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer*, pp. 55-61, July, 1984.

[12] Arvind and J. D. Brock, "Resource Managers in Functional Programming," *Journal of Parallel and Distributed Computing*, Vol.1, pp. 5-21, 1984.

[13] Mok, A. K., P. Amerasinghe, and M. Chen, "Synthesis of a Real-Time Message Passing System with Data-Driven Timing Constraints," *Proceedings of Real-Time Systems Symposium*, pp. 133-143, December 1987.

[14] Berzins, V. and Luqi, "Semantics of a Real-Time Language," *Proceedings of Real-Time Systems Symposium*, pp. 106-110, December 1988.

[15] Dennis, J., "First Version of a Data Flow Procedure Language," MIT TR-673, May, 1975.

A von Neumann/Dataflow Hybrid Approach To Real-Time Computing

Ragunathan Rajkumar

Robert A. Iannucci
and
Members of the Empire Project
K. Ekanadham, S. Gregor, K. Hiraki, M. Hale and P. Suhler

IBM Thomas J. Watson Research Center
Yorktown Heights

Abstract

As real-time systems become more demanding, multiprocessing and parallel processing techniques will be increasingly called upon to meet stringent timing requirements. This position paper outlines ongoing research on the development of a multiprocessor architecture which is a hybrid of the von Neumann and dataflow architectures, with built-in support for real-time systems. This research is being carried out by the Empire Project at IBM Research. A hybrid version of a dataflow architecture and a traditional von Neumann architecture has been proposed recently [7] and is also being studied by others [4, 5, 11]. The hybrid architecture supports built-in hardware mechanisms for parallel processing, synchronization, priority-based execution and context-switching. This set of features provided by the hybrid dataflow architecture is in particular attractive to real-time systems, where inter-process communication costs and the overhead of synchronizing parallel threads of execution can be prohibitive on traditional architectures. A rich set of primitives offered by the hybrid architecture opens up a fertile research ground for the use of appropriate compiler techniques, run-time systems and programming languages for real-time applications.

1. Introduction

Hard real-time systems require predictable timing behavior in the presence of stringent timing requirements. While fast hardware may be needed to handle the computing capacities required by real-time systems, predictable timing behavior can be guaranteed only by techniques such as the use of analyzable real-time scheduling algorithms [9, 12]. It is apparent that the real-time functions expected from embedded systems (such as the Space Station and the Mars Rover project) are becoming increasingly more demanding. This radical change in computing requirements needs to be supported by advances in hardware as well as in software. As a result, multiprocessors and distributed systems have found widespread use in recent years motivated by the potential speedup of applications run on them [10]. Multiprocessors, in particular, are attractive for parallelizing real-time tasks with stringent timing requirements. With parallel execution, these tasks can (potentially) be made to execute faster than on sequential uniprocessors. Nevertheless, in the context of hard real-time systems, this speedup becomes useful only if it is predictable.

2. A von Neumann/Dataflow Hybrid Architecture

The von Neumann architecture is better understood and used than any other architecture for uniprocessor systems. Unfortunately, the optimization techniques for von Neumann uniprocessor architectures do not always apply to multiprocessors. As the number of processors is scaled up, the latency to access global memory and the overhead of synchronizing tasks running on different processors become fundamental bottlenecks [3]. These two factors result in unpredictable latencies in memory and communications systems, and can also cause processor idleness in the form of pipeline bubbles. In specific, the latency cost to access memory is incurred on a per-instruction basis, but achieving synchronization on a per-instruction basis is impractical. In a dataflow architecture (such as the TTDA architecture [2]), an encoded dataflow graph represents the program, and machine instructions become self-sequencing. As a result, the dataflow architecture tolerates latency and synchronization costs naturally. However, intra-procedural communication is unnecessarily general in a dataflow machine. As pointed out in [7], it should not be necessary to create and match tokens for *every* instruction within a procedure body. Some scheduling can be done by the compiler, thereby saving corresponding run-time overhead.

The Empire Project at IBM Research is currently designing an architecture which is a hybrid of the von Neumann and dataflow architectures. The project goal is to marry these two architectures such that the advantages of both these approaches are exploited to provide scalable general-purpose parallelism. In addition, the architecture aims to provide built-in support for real-time systems. The distinctive features of the architecture are the following:

- The architecture supports multiple threads of computation within an invocation.
- The execution time of an instruction is independent of memory latency, giving rise to *split transactions*. The memory access delay can therefore be masked by other active instructions.
- The hardware supports synchronization on every word in memory leading to a very large (2^{48}) synchronization name space.
- The compiler-generated code calls for synchronization when and only when it is necessary.
- The machine language expresses the concepts of both implicit *and* explicit synchronization (thereby allowing partial ordering of totally ordered sequences of instructions).

3. The Compilation Target for the Hybrid Architecture

The compilation target for the hybrid architecture is as shown in Figure 1. Each instruction can refer to (up to) 2 source operands, and 1 destination operand. Each computation thread is represented by a single-word *continuation*. The continuation consists of a pointer to a *data frame* of 256 words, containing the data accessed by the corresponding thread, and indices into the data frame for the source operand(s) and the destination operand. The first word in the data frame (called the *Map entry*) points to a 256-word *code frame* containing the instructions executed by the thread. The instruction to be executed next by the thread is represented in the continuation by an index into the code frame. Instruction execution is normally sequential as on a von Neumann machine. If a thread tries (say) to read a location with unwritten data, it can be suspended and hardware automatically switches to other active threads. The suspended thread is reactivated (in hardware) when the corresponding data item is available. Thus, execution continues in dataflow fashion as well. Forks and joins of threads from the same invocation take place in a single cycle. Threads belonging to the same invocation share frames, but must run on the same processor.

The memory hierarchy consists of a main storage unit which is accessible from all processing elements. The main storage comprises of heap storage as well as code and data frames. Each processing element has a cache for a large number of data and code frames. Processor accesses to data and code are normally on the locally cached frames. Every frame is private to an invocation, and therefore does not pose any coherency problems. A frame can be locked such that when the frame cache fills up, the locked frame will not be cast out to global memory by the frame cache replacement policy.

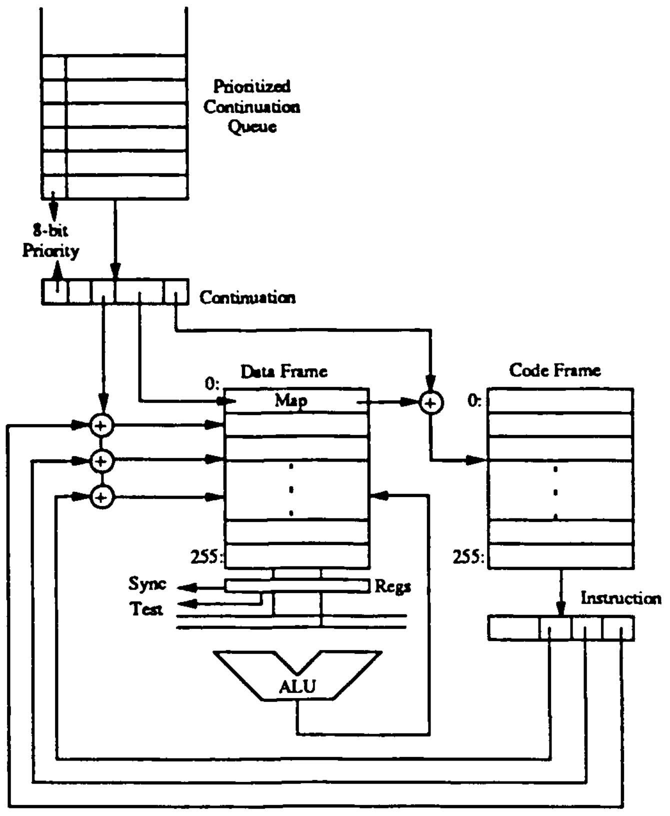


Figure 1: Compilation Target for Real-Time Applications

4. Support for Real-Time Systems

One of the goals of the Empire architecture is to provide built-in support for real-time systems. The primary motivation behind this goal is that the hybrid architecture provides low-cost synchronization functions which can be used effectively in real-time systems.

In this architecture, all continuations (threads) belonging to the same invocation have an associated priority represented by 8 bits. The continuation queue represents the traditional *ready queue* in operating systems, and is a hardware priority queue providing a very efficient means of building priority-driven real-time systems. An 8-bit priority representation was chosen for the reason that there is hardly any performance difference between using an infinite number of priority levels and 256 levels [8]. The architecture also supports the mechanisms of *get* and *set* priority, *create* and *kill* process, and *activate* and *suspend* process. These mechanisms are sufficient to build a priority-driven real-time system [14], and can be used to design an efficient yet predictable system based on analytical techniques such as that based on the rate-monotonic algorithm [9, 12].

Parallel and concurrent processing represent the implementation of an application as a set of cooperating tasks. Real-time applications, in particular, are characterized by the need for frequent communication between several tasks to coordinate various activities. A significant degree of speedup for real-time applications is therefore possible only if the synchronization and coordination of these tasks can be accomplished at low cost. The hybrid architecture provides a rich set of efficient synchronization mechanisms for real-time systems. State bits are associated with every word in memory, and allow synchronization to take place on every memory word on a per-instruction basis. Threads belonging to the same invocation synchronize (intra-procedural synchronization) within a cached frame, while inter-procedural synchronization takes place in the memory heap. In addition, the architecture supports producer/consumer state transitions. Transitions supported in hardware are "Multiple Producers, Single Consumer" with FIFO and LIFO queueing options, and the "One-time Producer, Multiple Consumers" synchronization mechanism.

The overhead of context-switching is often a concern in real-time systems. In the hybrid architecture, the state of any thread is represented by a single word, and a continuation is added to the ready queue in priority order by special-purpose hardware. As a result, context switching including queue insertion and deletion can be carried out within a few cycles.

The architecture explicitly supports process migration by means of mapping data frames to different PE's. This mechanism can be used to support mode-changes [13], fault-tolerance and dynamic reconfiguration in real-time systems. In addition, each processing element also has a real-time clock, and a countdown timer for implementing time-based operations.

5. Operating System and Language Issues

The new set of primitives offered by the hybrid architecture poses a host of interesting questions for real-time operating systems and languages. While the architecture itself may be novel, existing operating systems and languages can be supported well. However, given the primitives of the hybrid architecture, some tradeoffs traditionally employed in real-time systems must be reconsidered, and we expect that new approaches will become possible. For example, synchronization costs are often considered so prohibitive in real-time systems that extensive optimization and/or modifications are carried out on traditional architectures to reduce synchronization requirements. The overhead of the rendezvous synchronization mechanism in Ada [1] has proved to be a dominant factor in the limited use of the Ada tasking model in real-time systems. In addition, the scheduling of parallelizable tasks has not been studied extensively in the literature on real-time systems[1]. In contrast, the hybrid architecture provides efficient intra-procedural and inter-procedural synchronization, leading to cheap inter-process communication and efficient parallelization of real-time tasks. Furthermore, powerful synchronization mechanisms for the hybrid architecture can be built on top of the embedded support provided for multi-producer, single-consumer queues with FIFO and LIFO options. An example of such an abstraction is a multi-producer, multi-consumer discipline with priority-ordered producers/consumers using a serializing manager process.

Finally, the priority ordering mechanism almost always carried out in software can be carried out with extreme efficiency in hardware on the hybrid architecture. Context-switching can also take place in the duration of a few cycles, rather than in a few hundred cycles.

6. Current Status and Schedule

The Empire project at IBM Research is led by Dr. Bob Iannucci and consists of 6 other full-time researchers. The specification of the hybrid architecture is reaching completion, and a language called KUDOs has been designed for intermediate representation of parallelizable application programs.

[1]An analysis of parallelizable tasks from the scheduling point of view is presented in [6].

Languages that are planned for support include Fortran, Id, and Ada. An 8-processor hardware emulation engine with each processor emulating 8 logical processors is currently being prototyped, and will be completed by September 1990. A preliminary version of a software simulator for this prototype is currently operational. A 1024-processor prototype is tentatively planned for completion by the end of 1994. A software simulator of this final architecture is expected to be completed by the end of Summer 1990.

7. Acknowledgments

Credit for the architecture and software efforts are due to the Empire project comprising of Kattamuri Ekanadham, Steve Gregor, Mike Hale, Kei Hiraki, Bob Iannucci, Ragunathan Rajkumar and Paul Suhler.

References

[1] *Reference Manual for the Ada Programming Language*
U.S. Department of Defense, Washington, D.C., 1983.

[2] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali and R. Thomas.
The Tagged Token Dataflow Architecture.
August, 1983

[3] Arvind and R. A. Iannucci.
Two Fundamental Issues in Multiprocessing.
In Proceedings of DFVLR - Conference 1987 on *Parallel Processing in Science and Engineering.* Bonn-Bad Godesberg, June, 1987.

[4] Buehrer, R. and K. Ekanadham.
Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution.
IEEE Transactions on Computers C-36(12):1515-1522, December, 1987.

[5] Grafe,V.G., Hoch,J.E., and Davidson,G.S.
Eps'88:Combining the Best Features of von Neumann and Dataflow Computing.
Technical Report SAND-88-3128, Sandia National Laboratories, 1989.

[6] Han, C-C., Lin, K-J.
Scheduling Parallelizable Jobs on Multiprocessors.
Proceedings of the 10th IEEE Real-Time Systems Symposium , Dec., 1989.

[7] Iannucci, R. A.
Toward a Dataflow / von Neumann Hybrid Architecture.
In *Proceedings of the 15th Annual International Symposium on Computer Architecture.* June, 1988.

[8] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1 , May, 1986.

[9] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[10] Rajkumar, R., Sha, L., and Lehoczky J.P.
Real-Time Synchronization Protocols for Multiprocessors.
Proceedings of the IEEE Real-Time Systems Symposium :259-269, 1988.

[11] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T.
An Architecture of a Dataflow Single-Chip Processor.
In *Proceedings of the 16th Annual International Symposium on Computer Architecture.* June, 1989.

[12] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Task Scheduling in Distributed Real-Time Systems.
Proceedings of IEEE Industrial Electronics Conference , 1987.

[13] Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K.
Mode Changes in a Prioritized Preemptive Scheduling Environment.
The Real-Time Systems Journal , December, 1989.
Also available as a Technical Report, Software Engineering Institute.

[14] Sha, L. and Rajkumar, R.
Scheduling Mechanisms for Priority Driven Preemptive Scheduling.
2nd ACM International Workshop on Real-Time Ada Issues , June, 1988.

Fault Tolerance Approaches in Two Experimental Real-Time Systems: DREAM/LAN and MARS[1]

K.H. (Kane) Kim
Dept. of Electrical & Computer Engineering
University of California at Irvine
kane@balboa.eng.uci.edu

Andreas Damm
Institut für Technische Informatik
Technische Universität Wien, Austria
vmars!damm@tuvie.tuwien.ac.at

1 Introduction

Real-time systems can be exposed to a high number of environmental stresses and hazards that can lead to faults in the computer hardware. For example, high environmental and operating temperatures, short deviations in the power supply, vibrations, etc. can cause hardware faults. Various architectural measures can be taken to make real-time systems resistant against the occurrence of faults. Some of the major design issues concerning the dependability of real-time computer systems are the fault hypothesis, types of redundancy, recovery strategies, and the behavior of components upon the occurrence of faults.

Two significant attempts to experimentally evaluate some promising design choices have been under way at the authors' institutions for several years. The approaches to fault tolerance adopted in the two projects share some common characteristics (e.g., the type of redundancy) and yet reflect substantially different design philosophies (e.g., the degree of synchronism). Brief overviews of the two architectures will be given and the approaches taken to realize fault tolerance will be discussed. Finally, some possible extensions will be mentioned briefly.

2 Overview of the DREAM/LAN Model

The primary focus at UCI (University of California at Irvine) DREAM (Distributed Real-Time Ever Available Microcomputing) laboratory was to develop and evaluate system level fault tolerance techniques. However, observing predictable performance was a major guiding principle. The testbeds developed include tightly coupled multiprocessor systems as well as a local area network that is also used to simulate wide area networks [Kim89a]. From these systems the architectural model called DREAM/LAN is extrapolated. It basically consists of a number of cooperating fault-tolerant computing stations interconnected by a LAN.

In the DREAM/LAN model, computations are activated asynchronously. The execution of tasks is triggered by the arrival of messages generated internally or externally to a node. The sequence of task activations is determined dynamically under a deadline driven scheduling approach. The availability of knowledge on maximum execution times of real-time tasks is assumed. Tasks communicate by exchanging messages via the network.

[1]This work was supported in part by the University of California MICRO program and AT&T under Grant 89-148 and in part by the Austrian Science Foundation (Fonds zur Förderung der wissenschaftlichen Forschung) under contract J0350-PHY.

3 Overview of the Architecture of MARS

The fault-tolerant real-time system architecture MARS (Maintainable Real-Time System) has been developed at TUW (Technische Universität Wien) [KDK*89]. The basic building blocks of the MARS architecture are the so-called *MARS clusters*. Each cluster is composed of several *components* interconnected by a synchronous *real-time bus*. A component is a complete computer with its own copy of the MARS operating system kernel executing a set of real-time tasks. All components have access to a common *global time base* supported by distributed clocks with known synchronization accuracy [KO87].

MARS is a highly synchronous system architecture. All hard real-time tasks are activated according to a periodic schedule calculated off-line. The off-line resource scheduler plans not only the usage of each component's main processor but also the access to the cluster-wide real-time bus. Communication among tasks is realized by the exchange of periodic state-messages with a validity time. As soon as the validity of a message expires, the operating system kernel ensures that the message can no longer be read by application tasks. One characteristic feature of MARS is its highly deterministic timing behavior even under peak load and in the presence of faults.

4 Comparison of the Mechanisms Adopted for Realizing Fault Tolerance

Fault Hypothesis. The *fault hypothesis* defines the types of faults that are expected to occur and which the system is designed to cope with. Faults outside the fault hypothesis cannot necessarily be tolerated and might lead to system failures. Therefore, the fault hypothesis must be chosen to include all the faults that are expected to occur in a non-negligible fashion. The fault hypothesis of the DREAM/LAN model contains software faults as well as hardware faults. The current phase of the MARS project concentrates on transient and permanent hardware faults within the components and on the real-time bus. Software faults might be introduced into the fault hypothesis of MARS in the future.

Types of Redundancy Applied. Many kinds of redundancy like *hardware*, *software* (including information redundancy), or *time* redundancy can be applied to achieve fault tolerance. Depending on the point in time when redundant components are activated, we can distinguish between *static* (i.e., redundant components are active already during fault-free operation), *dynamic* (i.e., redundant components are passive during fault-free operation and become active only after the occurrence of a fault), and *hybrid* redundancy (i.e., static and dynamic redundancy are applied at the same time). Static redundancy (sometimes also called active redundancy) demonstrates a superior timing behavior compared to dynamic redundancy. An active-redundant node can provide its results immediately after a fault has occurred in the primary node, whereas a passive-redundant node under a dynamic redundancy scheme would need some time to perform the calculations that the primary node could not complete successfully. Therefore, static redundancy is favorable in real-time systems.

Static redundancy is the main approach employed in the DREAM/LAN model such that active-redundant nodes forming a computing station provide fast forward recovery. Within each node of a computing station software redundancy in the form of recovery blocks [Ran75] is incorporated. Redundant nodes use different try blocks as their primary blocks in parallel, thus transforming the dynamic software redundancy of classical recovery blocks into static software redundancy. The acceptance tests consisting of logic and time tests are identical in primary and backup nodes. Redundant nodes cooperate to send a single accepted result via the network. The resulting fault tolerance scheme is called the ***distributed recovery block (DRB) scheme*** [KW89]. Assuming that a computing station contains two nodes, three different failure cases can be distinguished upon the execution of the acceptance tests:

1. The primary node fails and the backup node passes. This situation will be recognized by the backup node when a notice from the primary does not arrive within the timeout period or when an explicit notice of an acceptance test failure comes from the primary node. The backup takes over the role of the primary and delivers its results to the successor node. The failed primary node tries to become the new backup node by executing its alternate try block.

2. The primary node passes and the backup node fails. The primary node is not disturbed at all and forwards its results. The backup node tries to recover by executing its alternate try block.

3. Both nodes fail. This might directly lead to a system failure in an application with highly stressful response time requirements. In any case, primary and backup nodes try to roll back and execute their alternate try blocks. If at least one of the nodes succeeds in passing the acceptance test, the station continues to serve the application.

The current version of the DRB scheme assumes the existence of only two nodes and two try blocks but conceptually the degree of redundancy can be increased to n hardware nodes and m versions of software. Under the DRB scheme, both software and hardware faults are treated uniformly to a large extent. The static hardware and software redundancy minimizes the delay for the delivery of results in case of a node failure.

MARS applies static hardware redundancy at the component level by running identical components in parallel. All redundant components receive the same input, execute the same operating system kernel and the same set of application tasks, and send their results as state-messages via the network. The filtering of redundant messages is done within the operating system kernel of the receiving component. Redundant components do not cooperate but run completely autonomously without knowing that (and how many) redundant components exist. A failed component does not inform any other component about its failure explicitly but remains silent. If one of a set of redundant components fails, the results produced by the remaining components are available without any delay, thus providing efficient forward recovery.

Time redundancy is incorporated in order to tolerate transient faults on the real-time bus. A PAR protocol using acknowledgments and retransmissions is avoided because of its unpredictable timing behavior. Instead, all messages sent are automatically duplicated by the operating system kernel and the duplicates are transmitted immediately after the original messages. Even during the time when only a non-redundant component is available, a transient fault on the real-time bus can thus be tolerated. Neither component faults nor transient faults on the real-time bus change the timing behavior of a MARS system upon their occurrence.

Recovery Mechanisms. *Backward recovery* is not well suited for real-time systems because of the significant time overhead involved in rollback and retry. Sometimes it is not even possible to roll back exactly to a previously established checkpoint because the state of the environment might have changed in the meantime. *Forward recovery* tries to compensate for the failed computation by establishing a new, valid state without rolling back to a previously stored state. Forward recovery is preferred in real-time systems for the obvious reason of recovery efficiency.

Both system models apply forward recovery at the computing station level which follows naturally from the static hardware redundancy applied. Different approaches are taken to recover a single node. In the DREAM/LAN, the failed node tries to recover by rolling back to the last checkpoint and executing the alternate try block to update its computational state and local real-time data-base (backward recovery). This causes the node to fall behind the primary node temporarily. If the computing station is not fully utilized, the failed node will catch up with the primary node sometime after it has finished its alternate try block.

Forward recovery in MARS is a very simple process provided by the static-redundant components. As long as two redundant components remain operational, the computing station can recover from a single hardware fault. However, since MARS components are assumed to be fail-stop, nodes shut down themselves whenever an error is detected and spare nodes have to be integrated even if the fault was only transient to reestablish the full fault tolerance capabilities of the system. The reintegration of components is in a conceptual study phase.

Fail-Stop Components. Components used to build fault-tolerant architectures can be categorized by their failure behavior. *Fail-stop* components decide by themselves whether their results are correct or not, i.e., they are self-checking. Extensive error checking is carried out within the component. Ideally, fail-stop components produce only correct results or no results at all. By running redundant fail-stop components in parallel, single faults affecting only one of a pair of fail-stop components can be tolerated. The overall complexity of fault-tolerant systems consisting of fail-stop components can be kept relatively low but much effort has to be placed into each node to achieve a self-checking coverage which is sufficiently high for critical real-time applications.

Both system models aim toward having fail-stop nodes as the basic building blocks of their architectures. In the DREAM/LAN model only information that has passed the acceptance test successfully is forwarded to the successor station. The acceptance test consists of a logic and a time acceptance test because missing a specified deadline for delivery of results is also considered to be a failure of a try block. This reflects the correctness criteria in real-time systems where results must be not only logically correct but also delivered in time to be valuable. Acceptance tests are the primary means of facilitating (approximate) fail-stop behavior of each node in the DREAM/LAN.

It is normally very expensive, if not infeasible to achieve perfect fail-stop behavior in any sizable computing component. In the case of using the DRB scheme, imperfect acceptance tests are a major source for propagated faults. Corrupt information that slip through an acceptance test can be propagated to other tasks executed on the same or on a different node. A *distributed conversation scheme* which provides fast forward recovery and supplements the DRB scheme by handling propagated faults is currently under investigation [Kim89b].

In MARS, a high self-checking coverage is sought by applying error-detection mechanisms at the operating system kernel level (plausibility tests, timing controls, etc.). Besides that, application tasks can be duplicated and their results compared. Currently, a new hardware component (single board computer) is under development that supports the operating system in achieving a high self-checking coverage by applying additional hardware error-detection mechanisms (like ECC codes, watchdog timers, control flow monitors, etc.). It is assumed that such a component can achieve a self-checking coverage above 99%.

5 Extensions

A number of research tasks aimed for extending the results achieved are in plan. For example, some of the major tasks planned in the DREAM laboratory are the tasks of evaluating distributed conversations with strict timing constraints and that of developing techniques for increasing the reconfigurability of fault-tolerant computing stations. Some of the future activities in the MARS project will include experimentally evaluating the self-checking coverage of the new hardware components currently under design as well as thorough evaluation of the dependability characteristics of a MARS system.

6 References

[KDK*89] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[Kim89a] K.H. Kim. An approach to experimental evaluation of real-time fault-tolerant distributed computing schemes. *IEEE Transactions on Software Engineering*, 15(6):715–725, June 1989.

[Kim89b] K.H. Kim. Approaches for system level fault tolerance in distributed real-time systems. In *4th Int. Conf. on Fault-Tolerant Computing Systems (ICFTCS4)*, pages 268–281, Baden-Baden, Germany, Sept. 1989.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[KW89] K.H. Kim and H.O. Welch. Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, May 1989.

[Ran75] B. Randell. System structure for software fault-tolerance. In *Proc. Int'l Conf. on Reliable Software*, pages 437–449, Los Angeles, April 1975.

Resource Management for Digital Audio and Video

David P. Anderson
University of California at Berkeley
EECS Dept., Computer Science Division
Berkeley, CA 94720

Ralf G. Herrtwich
International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704

January 30, 1990

Abstract. The *DASH resource model* is a workload and scheduling model that allows communicating peer entities to reserve the resources (such CPU and network bandwidth) necessary to achieve given delay and throughput objectives. The immediate goal of the model is to support digital audio and video in distributed systems. The model defines a parameterization of client workload, an abstract interface for hardware resources, and an end-to-end algorithm for negotiated resource reservation based on cost minimization.

1. Introduction

Audio and video (or *continuous media*) can greatly increase the effectiveness and range of a user interface. It will soon be possible to equip average workstations with the hardware to handle digital continuous media [1] and to connect these workstations by wide-area networks capable of handling continuous-media data [2]. This hardware base can support applications like video conferencing systems and the real-time display of video data stored on a remote file server [3].

Such applications transmit continuous data streams between hosts, and impose performance constraints (delay and throughput) on this transmission. The performance of current hardware (CPUs, networks, disks, etc.) is generally sufficient for continuous-media data. However, the scheduling policies used in hosts and gateways are designed primarily for fairness and simplicity. Therefore, especially during periods of heavy system load, the performance of a given connection may fail to meet the application's requirements. To remedy this situation, an approach to resource management with the following properties is needed:

- Resources (CPU, network, disk, etc.) that can potentially become bottlenecks must be scheduled in a way that allows "reservations" (with performance guarantees) to be made to individual clients.
- In making a reservation, clients must specify their workload. This is only possible if the workload is known when the reservation is made, *i.e.*, if the software generating the workload operates in a deterministic, time-invariant fashion.
- Since data may traverse resources on several hosts, a protocol for distributed resource reservation is needed. This protocol in turn requires a uniform interface to the various resources involved.

In this paper we describe the *DASH resource model* [4], a basis for resource reservation and scheduling in distributed systems designed to have these properties.

2. The DASH Resource Model

In the DASH resource model, the set of system components that handle continuous-media data is decomposed into a set of *resources*. A resource may correspond to a schedulable hardware device and its accompanying software driver. For example, a CPU and its scheduler can comprise a resource. Resources may also have a more complex structure: a local area network (which includes multiple interface devices, concurrent operation, and multiple scheduling mechanisms) might be treated as a single resource.

The DASH resource model assumes that work is assigned to resources in discrete units called *messages* (typically a segment of continuous-media data). Each message has a well-defined *arrival time* at which it is available for handling

This work was sponsored in part by the California MICRO program, AT&T Bell Laboratories, the Digital Equipment Corporation, Olivetti S.p.A., and the Hitachi Corporation.

by a resource, and *completion time* at which the handling is finished. The flow of continuous-media data is considered to consist of linear simplex message streams that pass through one or more resources. Data is generated by a *source* resource (a disk, digitizer, or compression unit), is then processed by a sequence of *handler resources* (networks, CPUs, etc.) and finally is consumed by a *sink resource* (disk, decompression unit, etc.). A message's completion time in one resource is its arrival time at the next resource. Many of these simplex data streams may exist concurrently, even within a single application. Therefore this scheme encompasses many continuous-media applications: playback of continuous media from disk, storage to disk, live conversations between human users, and so on.

2.1. Linear Bounded Arrival Processes

Each data stream flowing across an interface defines an *arrival process* into the downstream resource. The DASH resource model uses *linear bounded arrival processes (LBAPs)*, an abstraction introduced by Cruz [5]. An LBAP has the following parameters:

maximum message size	S_{max}	(bytes)
maximum message rate	R_{max}	(messages/second)
maximum burst size	B_{max}	(messages)

In any time interval of length t, the number of messages arriving at the interface may not exceed $B_{max} + t\,R_{max}$. The long-term data rate of the LBAP is $S_{max}R_{max}$ bytes per second. The burst parameter allows short-term violations of this rate constraint, modeling programs and devices that generate "bursts" of messages that would otherwise exceed the rate constraint.

We define a function $b(m)$ representing the logical "backlog" of the arrival process. This is the number of messages by which the arrival process is "ahead of schedule" (relative to its long-term rate) when message m arrives. The backlog is not necessarily the number of queued messages. $b(m)$ is defined by

$$b(m_0) = 0$$
$$b(m_i) = max(0,\ b(m_{i-1}) - (t_i - t_{i-1})R_{max} + 1)$$

where t_i is the arrival time of message m_i. Using $b(m)$, we define the *logical arrival time* $l(m)$ of a message m as

$$l(m_i) = t_i + b(m_i)\,/\,R_{max}$$

Intuitively, $l(m)$ is the time m would have arrived if the LBAP strictly obeyed its maximum message rate.

2.2. Sessions

The use of a resource by a particular data stream is called a *session*. A session represents a reservation of part of the capacity of the resource. Clients must establish sessions with all of the resources they need (using a scheme defined below) prior to sending data. As part of the reservation, the client must specify its workload; in return, the resource provides a bound on the delay it will impose.

Each session has associated sets of LBAP parameters for its input and/or output interfaces. A handler accepts LBAPs, producing output LBAPs. The client of the resource must enforce the input LBAP parameters; the scheduler of the resource must enforce the output parameters and delay bounds. We assume that handlers do not lose messages or modify their size, so the incoming and outgoing LBAP for handler resources must have the same values for S_{max} and R_{max}. On the other hand, incoming and outgoing LBAP may have different burst sizes. In addition to their LBAP specifications, handler sessions also have the following parameters:

maximum logical delay	L_{max}
minimum actual delay	A_{min}
maximum buffered delay	M_{max}

The *actual delay* of a message m in a handler resource is the time interval between its arrival at the input interface and its arrival at the output interface. The *logical delay* of m is the interval between the m's logical arrival time and its logical arrival time at the output interface. Logical delay, rather than actual delay, determines end-to-end delay bounds. The *buffered delay* is the portion of actual delay during which the message is stored in host memory. In resources such as wide-area networks, M_{max} will be smaller than L_{max} due to propagation time. A_{min} and M_{max} are used to calculate buffer space needs.

When data traverses a sequence of resources, the *basic sessions* within the resources are said to form an *end-to-end session*. An end-to-end session represents a unidirectional point-to-point communication path that traverses several resources, either within a single host or across a network. The output interface of each resource in an end-to-end session is the input interface of the next resource. Each resource must be prepared to handle the burst size generated by the previous resource. The *end-to-end logical delay* of a message is the interval between its logical arrival time at source output and its logical arrival time at the sink input. The maximum logical delay of an end-to-end session is the sum of the maximum logical delays of its handler resources.

2.3. Implementing the Resource Interface

The DASH resource model defines a uniform abstract interface to resources: clients request sessions, and the resource manager can grant or deny requests. The manager is responsible for honoring the delay and burst size limits for the requests it grants. If we restrict our attention to resources that consist of a single hardware device (e.g., a CPU), we see that the resource interface can be successfully implemented on top of a range of scheduling policies. Any policy for which an upper bound on delay can be derived from given input LBAP parameters can be used. For example, *round-robin*, *FIFO*, *rate-monotonic* and *earliest-deadline-first* scheduling all have this bounded-delay property. Many existing results in scheduling theory [6] can also be applied. If no guaranteed delay needs to be smaller than the interarrival time of messages on a session, a simple test for preemptive *rate-monotonic* scheduling of a singular resource would be

$$\sum_{s \in E} R_{\max}(s)\, T_{\max}(s) \le |E|\,(2^{1/|E|} - 1)$$

where E is the set of all established sessions (including the new one) and $T_{\max}$ is the maximum service time for each message [7]. Worst-case simulation provides a more general decision procedure [8].

For resources that encapsulate more than a single device, the management procedures are more complicated. For example, the sources of delay in an FDDI network (viewed as a single resource) include queueing, media access, and propagation, and many scheduling policies are possible. Establishing a session in such a resource may involve using network management protocols to reserve network bandwidth in "synchronous" or "isochronous" channels. The question of how to support sessions in such a resource is a subject of ongoing investigation.

2.4. An Economic Approach to Delay Allocation

It may be possible for a resource to guarantee a maximum delay anywhere within a certain range. The shorter the delay is, the more costly it is because the resource has less freedom to schedule its work items – and the fewer additional sessions it can support. To divide the delay between resources in an end-to-end session the DASH resource model takes an approach based on economics (this approach has also been used for problems such as routing and load-balancing [9]). When a client reserves a session with a resource, the resource makes reservations for the smallest possible maximum delay. In addition, the resource provides a *cost function* indicating, for each larger maximum delay, the associated cost to the client. The client may relax the maximum resource reservation to minimize cost.

Cost can either be real money, to be later billed to the client, or some metric reflecting the resource's current load. Typically, the cost function will not be static, but will be a function of the workload of a resource or parameters like the time of the day. For tractability, the DASH resource model requires that every cost function be 1) piecewise linear; 2) strictly monotonic decreasing, and 3) convex. The first value for which a cost is given is the smallest achievable maximum delay. We assume that at some point more delay will not lead to lower costs because the cost for buffering messages over the delay period will exceed the cost saved by the larger delay.

The cost of an end-to-end-session is the sum of the costs of its component sessions. Since we have defined cost functions to be piecewise linear convex functions, they can be combined by the following procedure: The segments of the functions are sorted in order of decreasing (more negative) slope. They are concatenated in this order, starting at the point which is the sum of the initial endpoints of the functions. The resulting function reflects the policy of returning excess delay in a way which minimizes total cost.

2.5. Buffer Reservation

The DASH resource model is designed to prevent message loss due to buffer overflow. For a given resource, this requires reserving enough buffer space to accommodate the input burst size plus messages being processed in the host. In bytes, this is given by the expression

$$S_{\max}(B_{\max} + R_{\max} M_{\max})$$

When several basic sessions are chained within a single host, buffer space to accommodate input bursts is only needed for the first session of the chain.

A second need for buffer space arises when the receiving end of an application must deliver messages at a constant rate to the output device (*e.g.*, audio or video converters). Suppose the first message of a stream arrives with minimum delay. If the application outputs the first message immediately, and the second arrives with maximum delay, there will be an unacceptable pause in the output between the two. Assuming that the source resource generates messages fast enough to maintain a nonzero backlog, this problem can be avoided as follows. The receiver waits until $R(\bar{L}_{max} - \bar{A}_{min})$ messages have been received (where $\bar{L}_{max}$ and $\bar{A}_{min}$ are the sums of L_{max} and A_{min} over all sessions in the end-to-end session), and then waits until the logical arrival time of the last of these messages. If O_{max} is the output burst size of the last session, the number of bytes needed for buffering is

$$S_{max}(O_{max} + R_{max}(\bar{L}_{max} - \bar{A}_{min}))$$

2.6. End-to-End Session Establishment

The DASH resource model defines an establishment protocol for end-to-end sessions. Using this protocol, the application's allowable end-to-end delay is divided between the resources, and burst sizes are established. The establishment protocol is carried out by *host resource managers* (HRMs). Initially, the HRM at the source host is given a client request that specifies the resources involved, the message size and rate, and the end-to-end logical delay requirements (a *target* and *maximum* value, denoted E_{target} and E_{max}). The protocol has two phases:

(1) The first phase traverses the hosts from the source to the sink. A *request* message is relayed between HRMs. The request message contains the data message size and rate, the client delays E_{target} and E_{max}, the burst size from the previous host, and the cumulative sums of L_{max} and A_{min}, and the cumulative cost functions. Maximum reservations are made for each resource, and corresponding buffer space is reserved.

(2) The second phase proceeds in the reverse direction. The receiving client evaluates the end-to-end session parameters and decides on a delay for the session. A *reply* message containing the remaining excess delay (see below) and the burst size into the next host is passed back towards the source. For each resource, the session parameters are relaxed appropriately. The delay may be increased and additional buffers may be reserved both for this purpose and to accommodate larger input bursts. Delays are relaxed only up to the amount of buffer space available.

Let E_{actual} be the actual end-to-end logical delay obtained in the first phase of the session establishment. If this delay is less than E_{target}, some *excess logical delay* E_{excess} defined as $E_{excess} = E_{target} - E_{actual}$ can be distributed among the resources. This should be done as economically as possible, *i.e.*, in a way which saves the largest amount of money. In the second phase of the protocol, each HRM hands the remaining excess delay to the previous one. Each HRM knows the outgoing accumulated cost function and the cost functions of all local resources. It shifts the outgoing cost function to the left, so that the first cost value is given for 0. From this cost function the segments from 0 to E_{excess} are examined. If any of these correspond to segments of local cost functions, the corresponding resources are relaxed by the time-extent of the segment, provided there is enough buffer space available. If E_{excess} lies in the middle of a segment, the amount of the relaxation is the part of the segment that lies to the left of E_{excess}. Any excess delay that is not returned to local resources is passed back to the previous host.

3. Conclusion

We have described the DASH resource model, a scheme to achieve guaranteed-performance communication in distributed systems. Other models, both statistical and deterministic, could offer the possibility of describing a broader range of traffic properties (for example, average message loss). The model we chose has the advantage of being simple and useful for deriving performance guarantees for a variety of scheduling disciplines. Our primary application area, continuous media, is well supported by our model.

The algorithms we have presented are conservative: worst-case assumptions are made, and during the establishment of an end-to-end session other establishment requests may be rejected needlessly because maximum reservations are made for each resource. This last problem can be avoided easily by blocking each new request until pending requests are completed. Optimistic approaches, similar to those in transaction processing, may be suitable for resource reservation in the real-time application domain.

Finally, the model deals with one-to-one communication only. This is sufficient in our current research where we apply the DASH resource model to IP-based communication in the Internet [10]. Since continuous-media systems will be used to a large extent for groupware applications like conferencing, multi-point connections are an important issue. Our future work will address this problem, extending the model of end-to-end sessions to include multicasting.

Acknowledgements

Martin Andrews, Robert Wahbe, Shin-Yuan Tzou and Ramesh Govindan contributed to the development of the DASH resource model. Thanks also to Domenico Ferrari for helpful comments and discussions.

References

1. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.
2. S. Newman, "The Communications Highway of the Future", *IEEE Communications Magazine 26*, 10 (October 1988), 45-50.
3. K. A. Frenkel, "The Next Generation of Interactive Technologies", *Comm. of the ACM 32*, 7 (July 1989), 872-881.
4. D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan and M. Andrews, "Support for Continuous Media in the DASH System", Technical Report No. UCB/CSD 89/537, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Oct. 1989.
5. R. L. Cruz, "A Calculus for Network Delay and a Note on Topologies of Interconnection Networks", Ph.D. Dissertation, Report no. UILU-ENG-87-2246, University of Illinois, July 1987.
6. S. Cheng, J. A. Stankovic and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems", in *Hard Real-Time Systems*, J. A. Stankovic and K. Ramamritham (editor), IEEE Computer Society, 1988, 150-173.
7. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *J. ACM 20*, 1 (1973), 47-61.
8. M. Andrews, "Guaranteed Performance for Continuous Media in a General Purpose Distributed System", Masters Thesis, UC Berkeley, Oct. 1989.
9. D. Ferguson, Y. Yemini and C. Nikolaou, "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems", *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.
10. D. P. Anderson, R. G. Herrtwich and C. Schaefer, "An Internet Protocol for Resource Reservation to Achieve Guaranteed-Performance Communication", Technical Report, International Computer Science Institute, Feb. 1990.

Timing Analysis of a Robot Motion-Planning Algorithm

Tom Bihari Dennis Pugh Tom Walliser Eric Ribble
Adaptive Machine Technologies
1218 Kinnear Road
Columbus, Ohio 43212
(614) 486-7741
amt@eagle.eng.ohio-state.edu

Prabha Gopinath
Philips Laboratories
North American Philips Corporation
Briarcliff Manor, NY 10510
psg@philabs.philips.com

April 13, 1990

1 Introduction

The axiom that "you can never have enough processing power" seems to be true in many application fields. However, as more processing power becomes available, it may make some existing problems trivial, while creating new uses for the power.

In the field of robotics, recent increases in processing power have made computer-controlled sensing and actuator servoing economically feasible. For many types of devices, the technological bottleneck for low-level sensing and servo-control is gradually shifting from computer technology to sensor, actuator, and materials technology. These same increases in processing power have opened the door for the possibility of executing robust high-level perception and planning algorithms in real time.

Temporal characteristics (e.g., execution times, deadlines) of high-level motion planning algorithms tend to have a different "flavor" than low-level servo-control algorithms. The execution times of many servo-control algorithms are more or less fixed, and the algorithms are required to execute periodically, on very short periods based on the mechanical characteristics of the device being controlled. We may be allowed to slightly vary the period or occasionally skip one or more periods, but major scheduling changes may have unpredictable results.

Motion-planning algorithms, however, may have temporal characteristics which depend on the current state of the device or the environment. Their execution times may increase as the environment becomes more difficult to traverse, for example. Because they are make use of environmental information, however, they may be able to adapt their execution to the environment. The following sections describe the temporal characteristics of a motion-planning algorithm used in the Adaptive Suspension Vehicle (ASV), a six-legged vehicle designed and built at AMT and the Ohio State University to study locomotion in rough terrain [1].

2 The Motion Planning Algorithm

The planner is driven by body velocity requests from the operator's joystick. The planner maintains a continuous vehicle velocity plan. The plan is made up of plan segments, which the planner periodically computes and adds to the plan. A plan segment consists of two portions: a normal portion containing requested body acceleration to be used for a certain duration into the future, followed by a safety portion containing contingency deceleration to zero body velocity. The safety portion of a plan segment is overwritten by the following plan segment, if the new plan segment is added to the plan by its deadline, the time at which the previous plan segment's normal portion expires. Since each validated plan ends with deceleration to zero velocity, the vehicle and its operator will remain safe even if a plan segment misses its deadline.

When computing a plan segment, the planner uses a super-real-time vehicle simulation [2] to verify the safety of all planned actions before they are executed by the actual vehicle. The simulation checks vehicle stability at small time increments called simulation intervals. If the simulation reveals accelerations which render the vehicle unstable, the simulation is repeated with reduced accelerations. Eventually, either the simulation is successful and the resulting sequence of body and leg states is appended to the previous plan, or the simulation fails and the previous plan remains in force.

Figure 1 shows a one-dimensional vehicle velocity plan (velocity is really a six-dimensional vector). The visible plan contains two plan segments, P1 and P2, each with a normal portion and a safety portion. Each portion is broken into several simulation intervals. If plan segment P2 is added to the plan before the actual time reaches the end of P1n, the vehicle continues with P2n, and so on. If plan segment P2 has not been computed by its deadline "deadline2", the plan follows P1s. Subsequent planning takes place based on the corresponding vehicle velocity.

3 Timing Analysis

The planner executes periodically. During each planning period, the planner generates one plan segment. The length of the plan segment may be fixed, or it may be allowed to vary (actually, only the length of the normal portion is directly selectable). The important constraint is that the planner must meet its deadline. That is, it must generate a plan segment and append it to the plan before the previous plan segment's normal portion runs out. The scheduling problem, as defined for this paper, is the problem of determining an appropriate length of plan segments, and corresponding planning period, given the available processing power.

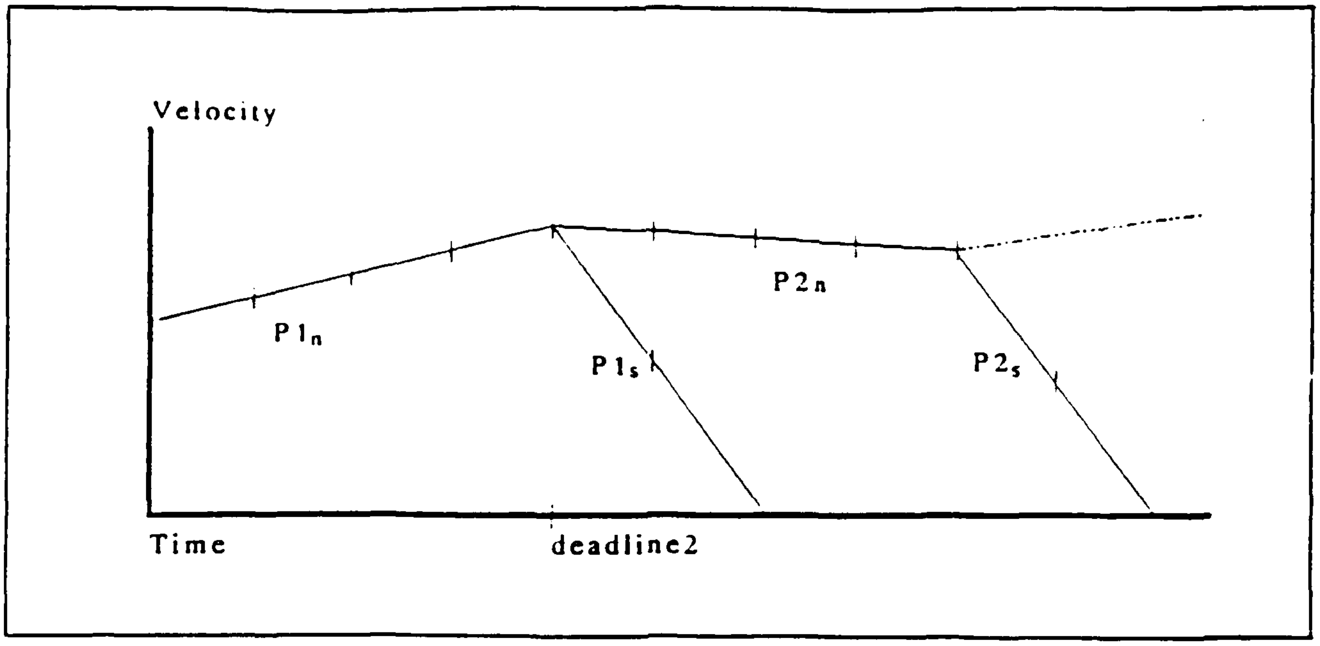


Figure 1: A Motion Plan

For a given processor, the amount of time required to calculate a plan segment is approximately:

$$DT_{cp} = \frac{V_{max} \times DT_{ci}}{A_d \times (DT_s - DT_{ci})} + DT_{ov}$$

where

DT_{cp}	= the amount of time to compute a plan segment;
V_{max}	= the vehicle's maximum allowable velocity;
A_d	= the acceleration during the safety "deceleration" portion of the plan segment;
DT_s	= the simulation interval;
DT_{ci}	= **the amount of time required to evaluate the vehicle dynamics during each simulation interval;**
DT_{ov}	= **a fixed amount of overhead time associated with computing any plan segment;**

The derivation of the cost function is not relevant to this discussion. However, the analysis raises several points:

The cost of planning is related to V_{max}. Since more can happen in a given planning period if the vehicle is moving rapidly, planning takes more time. However, the presence of V_{max} in the cost function could make it possible for scheduling algorithms to adapt performance to meet timing constraints. For example, the planner may be able to tell the vehicle: "Walk more slowly. I can't plan this fast.", much the way humans (presumably) operate.

The cost of planning is inversely related to A_d. "Slamming on the brakes" in response to a missed planning deadline is cheaper than coming to a slow, smooth stop. As with V_{max}, this could allow an adaptive scheduling algorithm to adjust the performance of the vehicle to available processing power.

When choosing the length of the normal portion of the plan segments, and therefore the planning period, we could schedule the planner frequently, thereby generating many, shorter, plan segments. This would have a higher overall cost because of the repeated DT_{ov} overhead. Furthermore, frequent, short executions of the planner are more likely to be affected by small scheduling disturbances, such as handling high-priority interrupts. This may result in plan segments missing their deadlines and activating the previous plan segment's safety portion, resulting in a bumpy ride.

At the other end of the spectrum, we could choose to generate longer normal portions of plan segments. However, because the time lag between a operator's command and its enactment by the vehicle is proportional to the length of a plan segment, long plan segments imply slow response to operator's commands. This may not be acceptable when maneuvering in close quarters, and does not allow the planner to replan rapidly in response to changing environmental conditions (such as foot slippage).

The ASV currently uses an adaptive scheduling algorithm. It starts with short plan segments, computed frequently. It monitors the planner, and if plan segments are completed uncomfortably close to their deadlines, it lengthens the normal portion of the plan segments (and increases the planning period).

4 Conclusion

The temporal characteristics of high-level, motion-planning algorithms tend to be less predictable than those of low-level, servo-control algorithms. This may make scheduling difficult.

Fortunately, robot motion-planning algorithms, by virtue of their level of interaction with the robot and environment, may be able to adapt their scheduling to the environment, and to adapt the behavior of the robot to the available processing power. This will allow robots to "intelligently" attempt to fulfill their duties, to the degree possible for the current situation.

References

[1] Thomas Bihari, Thomas Walliser, and Mark Patterson. Controlling the Adaptive Suspension Vehicle. *Computer*, 22(6):59–65, June 1989.

[2] W.J. Lee and D.E. Orin. The Kinematics of Motion Planning for Multilegged Vehicles Over Uneven Terrain. *IEEE Journal of Robotics and Automation*, (4):204–212, April 1988.

SOFTWARE DEVELOPMENT FOR HARD REAL-TIME SYSTEMS

Constance Heitmeyer and Bruce Labaw
Naval Research Laboratory
Washington, DC 20375

INTRODUCTION.

In [3], a *hard real-time (HRT)* system is defined as a system 'that must supply information within specified real-time limits'. Recently, the Naval Research Laboratory (NRL) initiated a new software engineering project to evaluate and extend techniques and tools for developing software for HRT systems. We are especially interested in the scaleability of the newer techniques and tools. In particular, to what extent do these help in the software development of real world systems with critical timing constraints? Our effort will build on work completed under an earlier NRL research project, called Software Cost Reduction (SCR), that applied software engineering principles to the reconstruction of the A-7E aircraft's Operational Flight Program (OFP).

In this paper, we briefly review the document-driven software development methodology used in SCR, describe the capabilities we seek in tools supporting HRT software development, and summarize our initial work with a prototype toolset called SARTOR, a product of ongoing research at the University of Texas (UT) [9,10,11]. Given its graphical language for specifying the software's functional and timing requirements and the automated assistance it provides for verifying that the resulting specification satisfies critical timing constraints, we believe that SARTOR represents a major step toward the next generation of tools supporting HRT software development.

SCR BACKGROUND

In SCR, a software development methodology was formulated whose major goal is to produce correct software that is easy to understand and easy to modify. With this methodology, the results of software development are captured in a series of formal, concise, carefully designed documents [13]. An initial document, the software requirements document, describes the required external behavior of the software. A second document, the module guide, describes the software structure, i.e., the software's decomposition into modules, where the criterion for module decomposition is the information-hiding principle. Later documents provide abstract specifications of the interfaces among the modules. Once the modules and their interfaces are specified, a series of module design documents specify the substructure of modules. These module design documents provide the basis for process decomposition, i.e., how a module is decomposed into a set of *sequential processes* [2,8], where each process contains operations that must be executed in a specific order.

These documents, examples of which are provided in [7,1,12], constitute a set of abstract descriptions of the software. An *abstract* description is one that many possible software implementations could satisfy. The most abstract document is the requirements document, which specifies only the software's externally visible behavior, postponing decisions about software structure, algorithms, and data representations to later stages of software development [5,6]. Later documents, such as the module guide, the interface specifications, and module design documents, are interpretations of the requirements document and as such must be consistent with it. These abstract documents lead eventually to the most concrete description of the software, namely, the executable software code.

CAPABILITIES OF A TOOLSET FOR DEVELOPING HRT SYSTEMS

We believe that the document-driven methodology described above should be encapsulated in a toolset supporting HRT software development. Summarized below are four capabilities we require in such a toolset. Two of these, verification of timing constraints and implementation on parallel processors, are of particular importance in developing HRT systems, while the the other two, checks on completeness and consistency and executability, are of more general importance.

Verification. A crucial requirement of a HRT system is that it satisfy critical timing requirements. We need assurance about the relative and absolute timing of certain critical events. To achieve such assurance, we seek tools that help developers formulate the required timing constraints and that help verify that the software specification satisfies these constraints.

Consistency and Completeness Checks. A major goal in SCR was to design documentation in a manner that makes inconsistencies and incompleteness apparent to the software developers. In [7], for example, output functions and mode transitions are presented using a tabular format, making some instances of inconsistency and incompleteness obvious. To aid software developers, we seek tools that automatically identify cases of inconsistency and incompleteness in a specification. Automated detection of some classes of incompleteness and inconsistency, e.g., missing or inconsistent table entries, is straightforward. Automatically detecting other, more complex cases, e.g., inconsistencies in the system's timing properties, is more difficult and requires more sophisticated tools.

Implementation on Parallel Processors. Reference [3] presents a design supporting sequential processes in HRT systems and their synchronization. Because it assumes global knowledge about the system state, this design is most useful when the software is implemented on a single processor. Distributing the system among several parallel processors requires an extension to this design, since, in a distributed architecture, each processor only has knowledge about its local state and must obtain knowledge about the state of other processors via external communication channels. Hence, we need a process model that allows implementation on distributed processors and a tool supporting that model. A tool that can automatically (or semiautomatically) produce a distributed implementation from a specification would be especially useful.

Executability. One approach to answering questions about HRT software requirements is to construct a prototype. Experience with a prototype can yield valuable information about the required functions and, especially important for a HRT system, about the system's timing properties. In addition, a prototype can be used to evaluate the specifications in the requirements document, i.e., to determine whether the requirements document accurately describes the desired external behavior. We believe that users are more likely to find errors by experimenting with a prototype than by poring over formal requirements specifications. Generating the prototype from the requirements document rather than from scratch should lead to greater assurance that the prototype is consistent with the requirements document. To accomplish this, we need a tool that provides assistance in translating a specification into a prototype.

SARTOR OVERVIEW

The SARTOR toolset includes a prototype tool called MODECHART which provides a graphical language for specifying software requirements [10, 11], additional tools that translate MODECHART into a form of first-order logic called Real-Time Logic (RTL) [9], and a prototype tool that provides automated assistance for verifying RTL assertions. Using the mode concept introduced in [7], MODECHART supports the partitioning of a system's state space into *modes*. Mode transitions are defined using predicates on state variables, events, and time intervals. RTL, which is designed for reasoning about a system's timing behavior [9], allows a system's behavior to be specified formally by a set of assertions. Often, a constraint on the system's logic that is desired by the software developers but not included in the software requirements may be expressed in RTL assertions; if the system has been described correctly, the assertion may be provable from the original MODECHART specification translated into RTL.

Recently, we evaluated existing requirements tools to determine whether any can support the capabilities described above. Although a few existing commercial tools, such as STATEMATE [4], provide limited support for some of these capabilities, e.g., completeness and consistency checks, no commercial tool supports the full set of capabilities. Our conclusion is that the SARTOR techniques and tools represent major progress toward a future production quality toolset providing these capabilities. The assistance that SARTOR provides to developers in proving timing assertions about the specification is a crucial step toward verifying that the software satisfies certain timing constraints. SARTOR should also prove helpful in checking a specification for inconsistency and incompleteness; while some straightforward checks can be done directly on a MODECHART specification, more complex checks will require translation of the MODECHART specification into a real-time logic and formal analysis of the logic. Further, a MODECHART specification could be the basis for automatic generation of a prototype. Finally, because it already supports both serial and parallel modes, refinement of a MODECHART specification into a specification that supports implementation on distributed processors is feasible.

INITIAL WORK ON SARTOR

In December, 1989, NRL installed a copy of MODECHART, obtained from UT, on a SUN 3/60. After experimenting with MODECHART and communicating with UT, NRL initiated several tasks involving MODECHART and other SARTOR tools, three of which we summarize below.

New Module Decomposition. Because both UT and NRL are working with SARTOR, we plan to do a module decomposition of the toolset with the goal of making SARTOR easier to change. This includes definition of abstract interfaces among the SARTOR tools and examination and possible redefinition of the module structure of individual tools. Because of limited resources, we anticipate only limited recoding. One possible change under consideration that will require some recoding is use of a commercial database system as the primary data storage medium for SARTOR.

Enhancements to User Interface. To the extent feasible, the user interface to the toolset should be easy to learn and use, should minimize the opportunity for errors, and should allow developers to focus on the problem at hand, i.e., describing the software requirements. To evaluate MODECHART as a specification language for software developers, we have been using MODECHART to describe the requirements of a complex HRT software package, the A-7 OFP, whose complete requirements are described in [7]. This initial use of MODECHART has suggested some possible enhancements to the MODECHART user interface. For example, currently, the tool only allows views of the requirements specification based on modecharts, a succinct form of state transition diagrams. We are currently exploring other views of the specification, e.g., tables, that would complement the modechart view.

Identification of Classes of Timing Properties. It is unlikely that an automated system can assist in the verification of all timing properties. Therefore, we are currently reviewing the A-7E requirements document to identify the classes of timing properties important in a complex HRT system. Identification of these timing classes will be provided to UT to suggest possible directions that future efforts to build verification tools might take.

TECHNICAL ISSUES

In our evaluation of MODECHART, a number of technical issues were identified as topics for future investigation. These are discussed briefly below.

How do we handle different models of time? The SCR model of mode transitions is different from that of MODECHART. MODECHART defines mode transitions as taking zero time. Events in MODECHART are associated with a particular time and can cause a chain of transitions if members of the chain are conditioned on an event's occurrence. In the SCR model of event triggering, only transitions waiting on an event before the occurrence of the event are triggered. This presents obvious difficulties in a distributed system. We plan to provide a formal method of translating between these two different models of time.

Should actions setting system states be explicit? At the highest level of software requirements specification, one may define state variables and use them without specifying when each state variable must be monitored. At a slightly more detailed level of specification, one must identify all actions that change the value of any state variable. At the first level, there is an assumption that the values of all state variables are updated as necessary. At the second level, the assumption is that no state variable is updated unless explicitly set by an action. Both levels are valid for specifying software requirements. Proofs about timing may require the second method.

ACKNOWLEDGMENTS

Many of the ideas presented in this paper resulted from discussions with P. Clements whose significant contributions are gratefully acknowledged. We also acknowledge Al Mok [illegible] his UT colleagues for developing the substantial conceptual foundation on which SARTOR is based, for building the toolset, and for allowing and supporting our experimentation with the toolset. We also thank our NRL colleagues, C. Brophy and S. Wilson.

REFERENCES

[1] Britton, K.H. and D. L. Parnas, 'A-7E software module guide," NRL Rep. 4702, Dec., 1981.

[2] Dijkstra, E.W., "Cooperating Sequential Processes," in F. Genuys (ed.), *Programming Languages*, Academic Press, New York, NY, 43-112, 1968.

[3] S.R. Faulk and D. L. Parnas, "On synchronization in hard real-time systems," *Commun. ACM 31,* 3 (March 1988), 274-287.

[4] D. Harel et al.,"STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Softw. Eng. SE-16,* 4, Apr. 1990.

[5] Heitmeyer, C.L. and J. McLean, "Abstract Requirements Specifications: A New Approach and its Application," *IEEE Trans. Softw. Eng. SE-9,* 5, Sept. 1983.

[6] Heninger, K.L., "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Softw. Eng. SE-6,* 1, Jan. 1980.

[7] Heninger, K.L. et al., "'Software requirements for the A-7E aircraft," NRL Rep. 3876, Nov., 1978.

[8] Hoare, C.A.R., "Communicating Sequential Processes," *Commun. ACM 17,* 10, (Aug. 1978), 666-677.

[9] Jahanian, F. and A. K. Mok, "Safety analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Softw. Eng. SE-12*, 9, Sep. 1980, 890-904.

[10] Jahanian, F. et al., "Semantics of MODECHART in Real Time Logic," *Proceedings, 21st Hawaii Intern. Conf. on System Sciences*, Jan. 5-8, 1988.

[11] Jahanian, F. and D.A. Stuart, "A Method for Verifying Properties of MODECHART Specifications," *Proceedings, Real-Time Systems Symposium*, Huntsville, AL, Dec., 1988.

[12] Parker, A. et al., "Abstract interface specifications for the A-7E device interface module," NRL Rep. 4385, Nov., 1980.

[13] D.L. Parnas and P.C. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Trans. on Software Eng. SE-12*, , Feb. 1986, 251-257.

Final Report to IEEE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903 (804) 924-7605

September 4, 1990

Ms. Anne Marie Kelly
Director of Conferences
IEEE Computer Society
1730 Massachusetts Avenue, NW
Washington, DC 20036-1903

Dear Ms. Kelly:

Enclosed please find a completed TMRF including the actual expenditures incurred by IEEE for the *Seventh IEEE Workshop on Real-Time Operating Systems and Software* which was held in Charlottesville on May 10-11, 1990.

Also included is a detailed analysis of the total expenditures for this conference which includes the matching funds in the amount of $10,000 which were received from the Office of Naval Research. In addition, the University of Virginia, School of Engineering and Applied Science, contributed the time for fiscal administration conference registration and management through the efforts of the School's conference director and the Department of Computer Science through secretarial support and my own effort.

A check in the amount of $ 265 to cover the 14% administrative costs on IEEE expenditures will be forwarded to you separately. If you have not received it within 30 days, or if you have any questions regarding the fiscal administration of this project, please contact Sandra Sullivan, Assistant Director, Engineering Academic Outreach at 804-924-3744.

Thanks for your help, please call me at 804-924-7605 if I can be of assistance to you.

Sincerely,

Robert P. Cook

Robert P. Cook
Computer Science Department

/ss

Enclosures:	TMRF
	Detailed Expenditures

c: Dr. K. Lin
Dr. S. Son
Dr. R. Lowry
Dr. G. Cahen
Ms. S. Sullivan

Seventh IEEE Workshop on Real-Time Operating Systems and Software
Actual Expenses to IEEE and ONR
May 9-11, 1990
Report 8/1/90

INCOME:

REGISTRATION FEES		IEEE	ONR	TOTAL
30 Members	@ $125	3750		
19 Late-Pay Members	@ $145	2755		
47 Total Members				
4 Non-Members	@ $150	600		
7 Late-Pay Non-Members	@ $180	1260		
11 Non-Members				
27 Students (10 UVA)	@ $ 50	1350		
87 Total Paid Participants				
7 Complimentary				
2 Conference Workers				
96 Total Attended				
Total Registration Income		**$9715**		
Matching Grant from Office of Naval Research			$10,000	
Total Income				**$19,715**

MEETING EXPENSES:

		IEEE	ONR	TOTAL
PROMOTION				
Conference Proceedings-Meeting		1,071		
Mailing, Fax, Federal Express			33.52	
**Call for Papers: gratis Cook		0		
**Final Program: gratis Cook				
**Advertising:			1,250.00	
Total Promotion:		**1,071**	**1,283.52**	**$2,354.52**
MEETING FUNCTIONS				
Meeting Facilities	Complimentary			
Audio-Visual Equipment			276.92	
Transportation	None			
Parking	Complimentary			
Total Meeting Functions:		**0**	**276.92**	**276.92**
CREDIT CARD PROCESSING FEE		272	0	272.00

		IEEE	ONR	TOTAL
ADMINISTRATIVE COSTS:				
Clerical @ !9.70/hour				
Type Badges (Materials included)	9 hrs.		61.80	
Registration/Mail/Telephone	24 hrs.	475	242.42	717.42
Registration/Desk-First day only,	4 hr.	79		
Conference Director-gratis	70 hr.			
Attendees List	1.5 hr.		29.70	
Fiscal Administration/Deposits/weekly reports			75.30	
Graphics @ $32.70 hr.				
Logo Cover	Included above			
Overheads				
Total Administrative Costs		**555**	**408.22**	**963.22**
TOTAL MEETING EXPENSES		**$1,897**	**$1,969.66**	**3,866.69**
Contingency -15% of Total Expense (Minimum $1000)				0
SOCIAL FUNCTION EXPENSES				
Reception:	$20.25x95x`1	$1,923		
Banquet: None		0		
Speaker's Breakfast	$22.32x25x2	558		
Lunch	$15.59x95x2	2,961		
Breaks:	$5.55x95x4	2,109		
Total Food and Function		**$ 7,553**	**$0**	**$ 7,553**
$80 per person				
TOTAL EXPENSE				
14% of Total Meeting Expense Paid by IEEE		**265**	**0**	**265.00**
PO for Newsletter		**0**	**8,030.34**	**8,030.34**
Total Expense		**$ 9,715**	**$10,000.00**	**$19,715.00**
TOTAL INCOME		**$ 9,715**	**$10,000**	**$19,715**

***School**
****Department**
*****ONR**

Seventh IEEE Workshop on Real-Time Operating Systems and Software CALL FOR PAPERS

May 10 - 11, 1990
University of Virginia
Charlottesville, VA USA

This workshop is the seventh in a continuing series of IEEE-sponsored workshops on real-time operating systems and software. The workshop has several goals:

to investigate advances in real-time operating systems;

to promote interaction among researchers and practitioners;

to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

Workshop attendees will explore the best current ideas on real-time software and operating systems. Position papers describing new ideas, promising approaches, and work in progress are considered particularly appropriate.

Possible topics of this workshop include:

Examples of real-time systems with challenging time constraints;

Real-time operating systems;
Real-time programming, requirements analysis, and specification;
Evaluation of real-time systems;
Real-time scheduling and resource management.

Prospective attendees should send 8 copies of a 5-page position paper to the Program Chair by February 1, 1990. The position paper should focus on insights and lessons gained from recent research and practical experience in real-time operating systems and software. Complete details regarding the workshop will be sent to all participants along with acceptance letters by March 15, 1990. A digest of accepted papers will be made available at the workshop. Attendance will be limited to 75 active workers in the field.

General Chair

Robert P. Cook
Dept. of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
USA
804-924-7605
cook@cs.virginia.edu

Program Chair

Kwei-Jay Lin
Dept. of Computer Science
1304 West Springfield Ave.
University of Illinois
Urbana, IL 61801
USA
217-333-1424
klin@cs.uiuc.edu

Program Committee

Wesley Chu, U.C.L.A.
Stuart Faulk, Software Productivity Consortium
Insup Lee, U. of Pennsylvania
Douglass Locke, IBM
Krithi Ramamritham, U. of Mass.
Kang G. Shin, U. of Michigan

Co-sponsored by
IEEE Computer Society
Technical Committee
on Real-Time Systems
Office of Naval Research

THE COMPUTER SOCIETY OF THE IEEE

ADVANCE REGISTRATION

Seventh IEEE Workshop on Real-Time Operating Systems and Software

May 10-11, 1990

UNIVERSITY OF VIRGINIA

Phone (804) 924-6268 Fax (804) 924-6270

NAME____

AFFILIATION____________________________________

ADDRESS______________________________________

CITY/STATE/ZIP__________________________________

ELECTRONIC MAIL ADDRESS__________________________

PHONE____

MEALS : Vegetarian?_______

ADVANCE REGISTRATION FEES:

	Before April 10	After April 10	
IEEE MEMBERS	**$125**	$145	IEEE Number:________
NON-MEMBERS	**$150**	$180	
STUDENT	**$ 50**	$ 50	

Proceedings and social functions are included in the price.

__PAYMENT ENCLOSED AMOUNT $__________

__INVOICE ME

__CHARGE MY MASTER CARD ____ CHARGE MY VISA CARD____

ACCOUNT NUMBER________________________

EXPIRATION DATE________________

__INVOICE MY COMPANY (purchase order #__________________)

EMPLOYER___________________________________

ADDRESS____________________________________

CITY/STATE/ZIP________________________________

PHONE ______________________________________

Please make your hotel reservations directly with the **OMNI-Charlottesville Hotel**, 235 W. Main Street, Charlottesville, VA 22901 **(804) 971-5500**. For special rate of **$79.00** (single or double) please tell them you will be attending the "IEEE/CS" conference. Reservations must be guaranteed by **April 9, 1990.**

Call **Enterprise Travel (800) 759-9437** and identify the "IEEE/CS" conference for travel arrangements and discounted airfares that are not available elsewhere, to Washington, DC or Charlottesville and transfer from the airport to the Omni. Enterprise agents will also be available to handle your travel needs during the conference.

After completing this form, return with check (U.S. Dollars) **made payable to *University of Virginia*** or Master Card Credit endorsement to:

Sandra Sullivan
University of Virginia
SEAS/Academic Outreach
Thornton Hall
Charlottesville, VA 22903

Seventh IEEE Workshop on Real-Time Operating Systems and Software
FINAL PROGRAM

Complimentary Breakfast (MR #8, Thursday's Speakers, Panelists, Session Chairs) 7:00 - 8:00

Opening Remarks (Salon C) 8:15 - 8:30
SESSION I: Operating Systems 8:30 - 10:30 (Chair: Andre van Tilborg)

Operating Systems Support for Adaptable Real-Time Systems
T. J. LeBlanc and E. P. Markatos, University of Rochester
CHAOS^{arc}: A Kernel for Predictable Programs in Dynamic Real-Time Systems
K. Schwan, H. Zhou and A. Gheith, Georgia Institute of Technology
Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel
L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic and G. Zlokapal, University of Massachusetts
LynxOS: UNIX Rewritten for Real-Time
I. M. Singh and M. Bunnell, Lynx Real-Time Systems, Inc.
BREAK: 10:30 - 11:00

PANEL: RT OS Standards: They are here, but are they good? 11:00 - 12:15
Hide Tokuda (Chair), Jane Liu, Doug Locke, Karen Gordon, R. Kuhn

LUNCH (Salon B) 12:30 - 1:30

SESSION II: Database and Concurrency Issues 1:30 - 3:30 (Chair: Wesley Chu)

Research in Time- and Error-Constrained Database Query Processing
G. Ozsoyoglu, Z. M. Ozsoyoglu and W. Hou, Case Western Reserve University
Scheduling Real-time Transactions in Distributed Database System
S. H. Son and J. Lee, University of Virginia
Preemption vs. Priority, and the Importance of Early Blocking
T. P. Baker, Florida State University
Supporting Real-Time Concurrency
V. Wolfe, S. Davidson and I. Lee, University of Pennsylvania
BREAK: 3:30 - 4:00

PANEL: Issues and Controversies in Guaranteeing Deadlines 4:00 - 5:30
Al Mok (Chair), Kang Shin, Pat Watson, Jack Stankovic, W.H. Spotz

Reception (Salon B) 6:00 - 8:00
Birds-of-a-feather (Junior Ballroom)
7:00 (ONR) Bob Fornaro, Proposed Testbed, North Carolina State
8:00 (ONR) Jim Smith, Proposed Test Cases, Office of Naval Research
Birds-of-a-feather (Meeting Room #8)
8:00 Still Open

COMPUTER SOCIETY
TECHNICAL MEETING REQUEST FORM
Request for Computer Society Approval of a
Conference, Symposium or Workshop

1. ABOUT THIS FORM

This form is to be used to (1) request that the Computer Society (CS) "sponsor", "co-sponsor", or "cooperate in" a technical meeting (Conference, Symposium, Workshop), or (2) file part of the final report for an approved technical meeting. For the request, complete the "estimates", for the final report, complete the "actuals".

To request sponsorship or co-sponsorship, please complete all sections of the form. To request cooperation, please complete sections 1 through 10, and M11 and T9.

For all meetings for which support is being sought for the first time, or for which there is reason to believe that supplementary information would speed the Computer Society review process, a supplementary information sheet should be attached which addresses the following points if they are not addressed elsewhere on the form: 1) Sponsors: if there is reason to believe that the Computer Society might not know a sponsoring or cooperating entity, for example, if it was recently formed, it would be best to include a brief description of the sponsor, its charter, its founders, its membership, and indicate if it is or is not a for-profit organization; 2) Technical Program: indicate what steps will be taken to assure the quality of the technical program, what will the paper review process be, etc. 3) Registration fees: what will registration fee structure be, will IEEE/CS members and members of other sponsoring and cooperating organizations be eligible for lowest registration rates (except for student rates and other special discounted rates, such as for retired members). 4) Publicity: will the meeting be publicized in such a way that IEEE members will have the opportunity to become aware of it. 5) CS members involvement: what will be the involvement of CS members in the technical and administrative operation of the meeting, 6) Schedule: adequate time (at least 9-12 months) should be allowed between proposal submission and meeting dates, 7) Attachments: the Draft Call for Papers and other relevant information should be enclosed, 8) Proceeds: a clear statement should be made indicating to whom proceeds are to go.

Please follow the guidelines in the Computer Society Conference Handbook: they are keyed to this form. Table V provides "Rule-of-Thumb Costs" to help complete the form.

Since a number of copies will be made of the completed forms, please use a typewriter or felt pen to make the entries, which should be made in US dollars. For meetings held outside the USA, indicate here the local currency (e.g., Swiss Francs) and the conversion rate used.

Local Currency __________ Conversion Rate __________ Local currency units per US dollar

This form will be valid until the end of 1987; after that time, contact the Director of Conferences for a more recent one.

Send completed form to: **The Computer Society, Director of Conferences, 1730 Massachusetts Avenue, N.W., Washington, D.C., 20036-1903.** (Phone 202-371-0101. TWX: 7108250437 IEEECOMPSO)

REQUEST FOR CS SPONSORSHIP ___X___ CO-SPONSORSHIP __________ COOPERATION __________

01/28/87

– PLEASE PRINT OR TYPE –

2. MEETING TITLE, DATES, LOCATION

Official Title of Meeting: Seventh IEEE Workshop on Real-Time Operating Systems and Software

Acronym:

Location (full address): University of Virginia, Thornton Hall, Charlottesville, VA 22903-2442

Housing Facilities (if different) ---

Dates: May 10-11, 1990

3. STATEMENT OF GENERAL CHAIR & FINANCE CHAIR

I have a copy of the Computer Society's "Conference Handbook" and I understand my responsibilities as outlined there.

This form including the budget has been prepared to the best of my ability and is complete and accurate. I understand th whenever it appears that the meeting may be in financial trouble, the Director of Conferences must be consulted.

I agree to provide the final report, to return the CS advance loan, if any, to return the CS share of the surplus funds, if ar and to close all accounts, all within four months after the meeting.

Further, I understand that all rights to this technical meeting are the property of and belong to the sponsoring entities.

General Chair Name Dr. Robert P. Cook IEEE/CS Member No. 7475361

Signature Date Sept. 26, 1989 Phones: Office (804)924-7605 Home (804)978-18

Address University of Virginia, Dept. of Computer Science, Thornton Hall
Charlottesville, VA 22903-2442

Finance Chair Name Dr. Sang Son IEEE/CS Member No. 2833598

Signature Date Sept. 26, 1989 Phones: Office (804) 924-7605 Home (804)296-87

Address University of Virginia, Dept. of Computer Science, Thornton Hall
Charlottesville, VA 22903-2442

3A. STATEMENT OF TECHNICAL COMMITTEE CHAIR (if sponsored by T/C)

I have approved this meeting as submitted.

I understand sponsorship involves a financial commitment by the T/C and that surpluses and losses will be apportioned as governed by correct C/S policy.

T/C #1 Name T/C Chair Signature

T/C #2 Name T/C Chair Signature

T/C #3 Name T/C Chair Signature

4. MEETING SCOPE, BENEFITS, ATTENDANCE
For first-time meetings, define scope and discuss overlap with approved CS mtgs

This is the Seventh IEEE Real-Time Operating Systems and Software Workshop

State benefits to society members: IEEE and ACM members will have an opportunity to contribute to the solution of real-time computing short falls in the Ada language. There will be technology transfer opportunities between industry and academia.

	ESTIMATED	ACTUAL
Attendance (from M11)	75	96
Sessions	12	7
No. of invited papers	3	0
No. of refereed papers submitted	90	57
No. of refereed papers accepted	30	18

5. SPONSORING & COOPERATING ENTITIES, & FINANCIAL COMMITMENT
List all entities, indicate if for-profit

Entity	Representatives Name & Telephone	% Financial Commitment	Commitment Obtained Prelim.	Commitment Obtained Final
Computer Society:	TC Real-Time Systems	100%		0%
TC (if applicable)	Dr. Robert P. Cook, (804)924-7605			
TC (if applicable)				
ACM:				
SIG (if applicable)	Office of Naval Research	$10,000		N/

6. SURPLUS & ADVANCE

	ESTIMATED	ACTUAL
Total income (from S1)	$19,790	9,715
Total expenses (from S2)	15,991	
Surplus (from S3)		

6. SURPLUS & ADVANCE (Continued)

Itemize expenditures requiring an advance:

Item	Amount
1.	$
2.	$
3.	$
4.	$
Total Advance Loan requested from all cosponsors	$
Total Advance Loan requested from Computer Society	$

Partial Advance Loans requested from CS and dates when needed:

$7,000 / 2 / 1 / 89 ______ / 4 / 1 / 89 ______ / 5 / 1 / 89 ______ / / /

7. ENCLOSURES

Enclose draft Call for Papers. If requesting sponsorship or co-sponsorship list below and enclose all contracts includin Hotel and Exhibits. and any other material relating to financial obligations.

8. STEERING COMMITTEE MEMBERS

	Name	Employer	Phone-Office	Phone-Home
Chair				

9. TECHNICAL MEETING COMMITTEE MEMBERS

	Name	Employer	Phone-Office	Phone-Home
General Chair	Dr. Robert P. Cook	UVA	(804) 924-7605	(804) 978-1892
Program Chair*	Dr. K. J. Lin	U of Illinois	(217) 333-1424	
Finance Chair	Dr. Sang Son	UVA	(804) 924-7605	(804) 296-8727
Tutorials Chair*	---			

*Please attach page showing mailing address.

9. TECHNICAL MEETING COMMITTEE MEMBERS (Continued)

	Name	Employer	Phone-Office	Phone-Home
Exhibits Chair	---			
Publicity Chair	Dr. Sang Son	UVA	(804)924-7605	(804)296-872
Registration Chair	Mrs. Sandra Sullivan	UVA	(804)924-6268	(804)985-739
Local Arrangement Chair	---			
Publications Chair	---			
Audio-Visuals Chair	---			
Contact	---			

10. PUBLICATIONS

	Yes	No
Proceedings	X	
Published by CS	X	
No. to be printed	200	
Sold by which Societies/Cos.	IEEE CS	
No. of pages	170	
Copyright assigned to IEEE	X	
If not, to whom		
Special issue of pub. planned	X	
If yes, name of publication	Real-Time Systems Newsletter	
Approved by pub. editor	X	

11. MILESTONES

Meeting Seventh IEEE Workshop on Real-Time Operating Systems Date May 10-11, 1990

Where a range if given, longer time is for Conference, shorter for Workshops. Asterisked items generally not applicable tc Workshops. Only items with "–" in front must have Name and Date entered to obtain CS approval.

BEFORE MEETING	Responsible Person(s)	Name	Date Due	Minimum Time Before Mtg.
–Define mtg., scope, etc.	Gen. Ch. & VPConf.	Cook	X	12-18 mos.
Sign hotel intent letter–	Local ArCh & DofC	Cook	X	10-14 M
Submit proposal for approval	Gen. Ch.	Cook	X	9-12 "
Appoint Committee	Gen. Ch.	Cook	X	9-12 "
–Hotel Contract Signed	Dir. of Con.	Cook	X	9-12 "
Committee Meeting	Gen. Ch.	N/A	X	9-12 "
–Exhibit Sales Contract	Exh. Ch&D of Conf.	N/A		*-12 "
Open bank account	Fin. Ch & Dir of Conf.	Son	XX	8-11 "
–Call for Papers[1]	Pgm. Ch.	Lin	X	8-11 "
–Finalize Publication Plans	Pgm.[illegible] Ch.	Lin	12/1/89	*-11 "
–Papers/Summaries from authors	Pgm. Ch.	N/A	3/1/90	5-11 "
–Advance announcement[1]	Pub. Ch.			5- 6 "
Tutorial spkrs. contracts	Tut. Ch. DofC.			*- 6 "
Pgm. Committee Meeting	Pgm. Ch.		3/31/90	4- 5 "
–Acceptance to authors	Pgm. Ch.	Lin		4- 5 "
–Author kits to authors	Pubs. Ch.	N/A		*- 5 "
Plan sales/membership booth/	Local Ar&Dof Press			*- 5 "
–Exhibit sales completed	Exh. Ch.	N/A		*- 5 "
Committee Meeting	Gen. Ch.	Cook	3/31/90	5- 5 "
–Advance pgm[1]	Pub. Ch.	Lin	4/15/90	3- 4 "
–Press Release	Pub. Ch.	N/A		3- 4 "
–Place magazine announce.[1]	Pub. Ch.	N/A		3- 4 "
–Final papers from authors	Pgm,[illegible] Ch.	Lin	3/1/90	*- 3 "

Footnotes:
(1) All ad copy required 6 weeks prior to month of issue. All advertising copy takes 7 8 weeks effort. Typesetting 5 7 days. Proofing 1 3 days. Printing 5 7 days Mailing Lists 3 weeks. Mailing 3 5 days: Postal Service 2 weeks.

11. MILESTONES (Continued)

BEFORE MEETING	Responsible Person(s)	Name	Date Due	Minimum Time Before Mtg.
–Magazine ads appear[1]	Pub. Ch.	N/A		8-12 wks.
Audio-visuals quality rev. Aud.	Vis. Ch.			6-10 "
–Proceedings to printer	Pub. Ch.	Son	4/1/90	•-10 "
Final session room assign.	Local Ar. Ch.	Cook	4/15/90	•- 8 "
Tutorial notes to D of Conf.	Tut. Ch.	N/A		•- 6 wks.
Session signs available	Local Ar. Ch.	N/A		•- 5 "
–Final program[1]	Pgm. ~~Pub.~~ Ch.	Lin	4/5/90	•- 5 "
–Advance registration closes	Reg. Ch.	Sullivan	4/15/90	4- 4 "
–Final program delivered	Reg. Ch.	Sullivan	5/10/90	•- 3 "
Badges/ribbons available	Reg. Ch.	Sullivan	5/10/90	3- 3 "
Hotel food planning quantity	Local Ar.	Cook	X	2- 2 "
–Proceedings delivered to site	Pub. Ch.	Son	5/8/90	•- 3 days
Hotel food quantity guaranteed	Local Ar.	Cook	4/15/90	at Conf.

AFTER MEETING				
Committee debriefing	Gen. Ch.			Day after
Recommend committee awards	Gen. Ch.			4- 4 wks.
–Interim Report & Advance Return	Gen. & Fin. Ch.			2- 2 mos.
–Final report & monies	Gen. & Fin. Ch.	Cook	9/1/89	4- 4 mos.

Footnotes:
(1) All ad copy required 6 weeks prior to month of issue. All advertising copy takes 7-8 weeks effort. Typesetting 5-7 days; Proofing 1-3 days; Printing 5-7 days; Mailing Lists 3 weeks; Mailing 3-5 days; Postal Service 2 weeks.

X = already completed
N/A = not applicable

12. BUDGET

MEETING EXPENSES
(Exclude tutorials and exhibits)

	Estimate	Actual
M1 Advertising (including printing, handling, mailing)		
Complete attached advertising worksheet.		
(a) Call-for-Papers (019)	$ 4,000	$ 0
(b) Announcement (021)	$	$
(c) Advance Program (020)	$	$
(d) Posters (022)	$	$
(e) Other (specify) (017)	$	$
	$	$
Subtotal for advertising	$	$
(e) Tutorial expense (if tutorial advertising is not budgeted separately, subtract 20% and add it to T1)	$	$
M1 Total Advertising	$ 4,000	$ 0
Adv. cost as % of total tm cost (M1 / M10)	$ 26%	$ 0
M2 Committee Expenses		
(a) Secretary (851) No. hours ____ X $/hr. ____ =	$	$
(b) Telephone (830)	$	$
(c) Postage (570)	$	$
(d) Committee Travel (871)	$ 1,000	$ 0
(e) Reproduction (190)	$	$
(f) Compmail +[2] (830)	$	$
(g) Other (specify) (517)	$	$
	$	$
M2 Total Committee Expenses	$ 1,000	$ 0

Footnotes
[2] Technical meetings with budgets over $30K should include funds for a Compmail account for key members of the Steering and Tech. Meeting Committees.

12. BUDGET (Continued)

REMARKS (c) Program Committee will meet at Dulles airport, near Washington, D.C. on March 31, 1990 to select program.

M3 Operating Expenses

	Estimate	Actual
(a) Advance Registration		
(1) By Computer Society (761) (70% est. attendance X C_1 from Table IV)	$ 0	$ 0
(2) or by other means (show computation) (768)		
24 hrs. x $19.70 mail/telephone/registration	$ 473	$ 475
(b) On-site registration[3] (762) one person, one half - NCOA 4 x 19.70	$ 158	$ 79
(c) Guards (771)	$ --	$
(d) Gratuities, awards, attendee travel[1] (030)	$ --	$
(e) Keynote and special addresses[1] (875)	$1,000	$ 0
(f) Audio Visuals – Labor & Equipment (719) 8 x 10 screen	$ 70	$
(g) Typewriters and other equipment (713)	$ --	$
(h) Final program (artwork and printing) (600)	$ 0	$
(i) Proceedings for attendees[4] (615)	$	$
No. copies 90 x $/page .07 x 170 No. pages	$1 071	$ 1,071
(j) Signs (meeting rooms, other) (795)	$ 0	$
(k) Meeting space rental (715)	$ 212	$ 0
M3 Total operating expenses	$2,984	$ 1,626

Footnotes
(1) These items must be explained under Remarks.
(3) If done by CS Staff, enter C, from Table IV per staff member plus travel expenses.
(4) Cost per page if done by CS Press is given on Table V.

12. BUDGET (Continued)

M4 Other technical meeting expenses

Include and describe any expense not identified on previous pages.

Bank charges (070)	$	$
Rebates (670)	$	$
Bad debts (650)	$	$
Insurance (396)	$	$
Other (511) Credit card bank processing fee.	$ 332	$ 272
M4 Total other expenses	$ 332	$ 272
M5 Meeting Expenses Subtotal Add 1 M1, M2, M3, M4	$ 8,316	$ 1,897
M6 Contingency (180) Enter 5 to 15% of line M5 ($1000 minimum) ______ %	$ 1,000	$ 0
M7 Computer Society Administrative Services[5] (780) Enter 14% of line M5	$ 1,165	$ 265

Footnotes:
(5) This is a mandatory entry for all meetings; it helps recover expenses incurred by the Computer Society for all technical meetings. For co-sponsored meetings, this expense will not be charged to the meeting but will be taken from the Computer Society share of the surplus.

REMARKS

12. BUDGET (Continued)

M8 Social Functions

(a) Coffee, pastries, etc., between sessions (472) No. breaks 4 X No. people 95 X S/person 5.72	$ 1,575	$ 2,110
(b) Luncheons (471) No. Luncheons 2 X No. people 95 X S/person 15.59	$ 1,875	$ 2,962
(c) Receptions (473) No. Receptions 1 X No. people 95 X S/person 20.24	$ 1,600	$ 1,923
(d) Banquets (474) No. Banquets ____ X No. people ____ X S/person ____	$ 0	$ 0
(e) Speakers Hospitality (475) No. people 25 X S/person 22.32 x 2 days	$ 460	$ 558
(f) Transportation (courtesy bus, etc.) (861)	$ N/A	$ 0
(g) Other social functions (specify) (476)	$	$ 0

M8: above includes gratuities, taxes, and any related service fees.

M8 Total Social Function Expenses	$ 5,510	$ 7,553
Social cost per attendee	$ 74	$ 80

M9 Services from Computer Society Staff

Use Table IV to find the charge for any service desired and enter the amount below. N/A

SERVICE	YES	CHARGE Estimated	Actual
Call for Papers Artwork	___	Include in M1 & Worksheet	
Announcement Artwork	___	Include in M1 & Worksheet	
Advance Program Artwork	___	Include in M1 & Worksheet	
Final Program Artwork	___	Include in M3 & Worksheet	

12. BUDGET (Continued)

M9 Services from Computer Society Staff (Continued)

SERVICE	YES	CHARGE Estimated	CHARGE Actual
Mail Call for Papers	____	Include in M1	
Mail Announcement	____	Include in M1	
Mail Advance Program	____	Include in M1	
Prepare Budget (763)	____	$________	$________
Treasurer's Service (764)	____	$________	$________
Advance Registration[6]	____	Include in M3	
On-Site Registration	____	Include in M3	
Hotel Negotiations (765)	____	$________	$________
Other Negotiations (specify) (766)	____	$________	$________
________________			
Prepare & Place Press Releases	____	Include in M1 & Worksheet	
Prepare Advertisements	____	Include in M1 & Worksheet	
On-site Publications Sales	____	No charge	
On-site Membership Booth	____	No charge	
Proceedings Publication	____	Include in M3	
Other services of the East or West Coast offices (specify) (767)			
________________	____	$________	$________
________________	____	$________	$________
________________	____	$________	$________
M9 TOTAL		$________	$________
M10 Total Technical Meeting Expenses Add M5, M6, M7, M8, M9		$ 15,991	$________

Footnotes:
(6) The Director of Conferences should be treasurer for all conferences requesting advance registration processing by the C. S. A finance chair should still be appointed to the conference committee in this case.

MEETING INCOME
(Exclude tutorials & exhibits)

	Estimated						Actual					
M11 Registration												
Advance Registration												
Members[7]	60	@ $	125	= $	7,500		30	@ $	125	= $	3,750	
Non-members[8]	5	@ $	150	= $	750		4	@ $	150	= $	600	
Full-time student members[9]	0	@ $		= $			27	@ $	50	= $	1,350	
Other[10] (specify below)	0	@ $		= $				@ $		= $		
Late / On-site Registration[11]												
Members[7]	7	@ $	145	= $	1,015		19	@ $	145	= $	2,755	
Non-members[8]	3	@ $	175	= $	525		7	@ $	180	= $	1,260	
Full-time student members[9]		@ $		= $				@ $		= $		
Complimentary[10]	9											
Other[10] (specify below)		@ $		= $				@ $		= $		
Total attendance[12]	75						96					
Last year's paid meeting attendance	75											

	Estimated	Actual
M11 Total Registration Income	$ 9,790	$ 9,715
M12 Other Income (specify)		
Pub sales (300)	$ 0	$
Interest (140)	$ 0	$
Other (706) Grant from Office of Naval Research	$ 10,000	$ N/A
M13 Total Income (M11 plus M12)	$ 19,790	$ 9,715

Remarks ______________________________

Footnotes
(7) Members of the Computer Society, IEEE, co-sponsoring or cooperating entities
(8) Non member rates should be 25 30% higher than member rate.
(9) Student rates usually are for sessions only and do not include Proceedings or social functions. if otherwise, please indicate under Remarks.
(10) Specify under Remarks, who will receive complimentary or special rates, and indicate if the rate includes a copy of the Proceedings and attendance at social functions. If committee members, speakers, session chairs, etc., will receive complimentary or special rates, they must be listed here or they must pay the appropriate member or non member rate. Use discretion. Retired members are entitled to reduced rates. Special combination rates offering discounts for attending two functions must be shown ie. Conf. + 1 Tutorial, Conf. + 2 Tutorials or 2 Tutorials etc.
(11) Late On-site Registration rates should be at least 20% higher than advance.
(12) An estimate more than 10% higher than last year's actual should be explained.

TUTORIAL EXPENSES

T1 Advertising (including printing, handling, mailing). Complete attached worksheet. If tutorial advertising is not budgeted separately, enter amount from M1.e

	Estimated	Actual
T1 Total tutorial advertising (026)	$ 0	$ 0

T2 Operating Expenses

	Estimated	Actual
(a) On-site registration[13] (768)	$	$
(b) Guards (772)	$	$
(c) Gratuities (031)	$	$
(d) Audio Visuals – Labor & Equipment (717)	$	$
(e) Typewriters and other equipment (720)	$	$
(f) Texts (613) No. copies ______ X $/copy[14] ______	$	$
(g) Notes (614) No. copies ______ X $/copy[14] ______	$	$
(h) Signs (798)	$	$
(i) Speaker fees and travel expenses		
No. of tutorials ______ No. of days ______		
No. of full-day speakers ______ x Rate[14] ______ (637)	$	$
No. of half-day speakers ______ x Rate[14] ______ (637)	$	$
No. of spkrs ______ X travel exp/spkr ______ (872)	$	$
(j) Meeting space rental (721)	$	$
T2 Total operating expenses	$ 0	$

T3 Other Tutorial Expenses (512)
Include and describe any expense not identified above

	Estimated	Actual
______	$	$
______	$	$
______	$	$
T3 Total other tutorial expenses	$ 0	$ 0

Footnotes
(13) If this will be done by Computer Society staff, enter C, from Table IV plus travel expenses.
(14) Refer to Table V.

TUTORIAL EXPENSES (Continued)

		Estimated	Actual
T4	Tutorial Expenses Subtotal Add T1, T2, T3	$______	$______
T5	Contingency (181) Enter 5 to 15% of T4 ______%	$______	$______
T6	Computer Society Administrative Services[5] (781) Enter 14% of line T4	$______	$______
T7	Social Functions		
	(a) Coffee, pastries, etc. between sessions (481) No. breaks ______ X No. people ______ X $/person ______ =	$______	$______
	(b) Luncheons (482) No. luncheons ______ X No. people ______ X $/person ______ =	$______	$______
	(c) Receptions (483) No. Recept. ______ X No. people ______ X $/person ______ =	$______	$______
	(d) Other social function expenses (specify) (484) ______________________________	$______	$______
	T7 Total social function expenses	$ 0	$______
T8	Total Tutorial Expenses Add T4, T5, T6, T7	$ 0	$ 0

REMARKS

____

____

____

____

____

TUTORIAL INCOME

	Estimated	Actual
T9 Registration Income		
Advance Registration (440)		
Members[7]	____ @ $____ = $____	____ @ $____ = $____
Non-members[8]	____ @ $____ = $____	____ @ $____ = $____
Full-time student members[9]	____ @ $____ = $____	____ @ $____ = $____
Other[10] (specify below)	____ @ $____ = $____	____ @ $____ = $____
Late / On-site Registration[11] (442)		
Members[7]	____ @ $____ = $____	____ @ $____ = $____
Non-members[8]	____ @ $____ = $____	____ @ $____ = $____
Full-time student members[9]	____ @ $____ = $____	____ @ $____ = $____
Other[10] (specify below)	____ @ $____ = $____	____ @ $____ = $____
T9 Total Registration Income	$ 0	$
T10 Other Income (specify) (709)		
____	$	$
____	$	$
T10 Total Other Income	$ 0	$ 0
T11 Total Tutorial Income (T9 plus T10)	$ 0	$ 0

REMARKS

Footnotes

(7) Members of the Computer Society, IEEE, co-sponsoring or cooperating entities

(8) Non-member rates should be 25-50% higher than member rate.

(9) Student rates usually are for sessions only and do not include Proceedings or social functions: if otherwise, please indicate under Remarks.

(10) Specify under Remarks, who will receive complimentary or special rates, and indicate if the rate includes a copy of the Proceedings and attendance at social functions. If committee members, speakers, session chairs, etc., will receive complimentary or special rates, they must be listed here or they must pay the appropriate member or non-member rate. Use discretion. Retired members are entitled to reduced rates. Special combination rates offering discounts for attending two functions must be shown ie. Conf. + 1 Tutorial, Conf. + 2 Tutorials or 2 Tutorials etc.

(11) Late On-site Registration rates should be at least 20% higher than advance.

EXHIBIT EXPENSES

	Estimated	Actual
E1 Advertising (including printing, handling, mailing) (015)		
Complete attached advertising worksheet. If exhibit advertising is not budgeted separately, enter a pro-rated amount of the advertising budget for the meeting.		
E1 Total Exhibit Advertising	$ 0	$
E2 Operating Expenses		
(a) Space Rental (714)	$	$
(b) Management Fee (635)	$	$
(c) Security (773)	$	$
(d) Insurance (397)	$	$
(e) Busing (862)	$	$
(f) Drayage (712)	$	$
(g) Other expenses (specify) (516)	$	$
	$	$
	$	$
E2 Total Exhibit Operating Expenses	$ 0	$ 0
E3 Exhibit Expenses Subtotal Add E1, E2	$ 0	$ 0
E4 Contingency Enter 5 to 15% of E3. . ______% (182)	$ 0	$ 0
E5 Computer Society Administrative Services[5] (782) Enter 14% of E3	$ 0	$ 0
E6 Total Exhibit Expenses Add E2, E3, E4, E5	$ 0	$ 0

Footnotes:
(5) This is a mandatory entry for all meetings; it helps recover expenses incurred by the Computer Society for all technical meetings. For co-sponsored meetings, this expense will not be charged to the meeting but will be taken from the Computer Society share of the surplus.

EXHIBIT EXPENSES (Continued)

EXHIBIT INCOME

	Estimated	Actual
E7 Exhibitor Fee Income		
No. Exhibitors ______ X $/Exhibitor ______ (410) .... =		
or No. Booths ______ X $/Booth ______ (410) .... =	$______	$______
E8 Other Exhibit Income (specify) (708)	$______	$______
______	$______	$______
______	$______	$______
E9 Total Exhibit Income (E7 plus E8)	$ 0	$______

REMARKS
1 Describe exhibit facilities, state $/sq. ft. sales price of booth space, attach copy of any contracts, etc.

______

______

______

S BUDGET SUMMARY

	Estimated	Actual
S1 Income		
M13 Total Meeting Income	$ 19,790	$ 9,715
T11 Total Tutorial Income	$ 0	$ 0
E9 Total Exhibit Income	$ 0	$ 0
S1 TOTAL INCOME	$ 19,790	$ 9,715
S2 Expenses		
M10 Total Meeting Expense	$ 15,991	$ 9,715
T8 Total Tutorial Expense	$ 0	$ 0
E6 Total Exhibit Expense	$ 0	$ 0
S2 TOTAL EXPENSES	$ 15,991	$ 9,715
S3 Surplus (S1 minus S2)	$ 3,799	$ 0

ADVERTISING WORKSHEET

13. The costs given below represent approximate costs of services available through the Washington office, and are mean to be guidelines only. Actual charges may vary.

13.1 Call for Papers

Artwork & Typesetting – Includes handling charge
(No. of pages ______ X $350) .... $______

Printing: ______________ Qty. .... $______
See Table I

Mailing
Postage (Bulk) @ 9.0¢ X______________ Qty. .... $______

Postage (First Class) @ 22¢ (25¢ w/envelope) X______________ Qty. .... $______

Mailing Labels (see Table II) @ 4.5¢ X______________ Qty. .... $______

Labor @ 3¢ per copy X______________ Qty. .... $______

Periodic advertisement (List Names & Issue Dates) .... $______
(See Table III)

Press releases preparation & mailing @ $150/release X______________ Qty. .... $______

Total, Call for Papers .... $______

13.2 Announcement

Artwork & Typesetting – Includes handling charge
(No. of pages ______ X $350) .... $______

Printing: ______________ Qty. .... $______
See Table I

Mailing
Postage (Bulk) @ 9.0¢ X______________ Qty. .... $______

Postage (First Class) @ 22¢ (25¢ w/envelope) X______________ Qty. .... $______

Mailing Labels (see Table II) @ 4.5¢ X______________ Qty. .... $______

Labor @ 3¢ per copy X______________ Qty. .... $______

Periodic advertisement (List Names & Issue Dates) .... $______
(See Table III)

Press releases preparation & mailing @ $150/release X______________ Qty. .... $______

Total, Announcement .... $______

ADVERTISING WORKSHEET (Continued)

13.3 Advance Program

Artwork & Typesetting – Includes handling charge
(No. of pages _____ X $350) .... $______

Printing: 2 Qty. full page ads $1500 ea. .... $ 3,000
See Table I

Mailing (See Table II)
Postage (Bulk) @ 9.0¢ X______ Qty. .... $______

Postage (First Class) (22¢, 4 pages; 37¢ m.t 4; + 3¢/envelope) X______ Qty. 4000 .... $ 1,000

Mailing Labels (see Table II) @ 4.5¢ X______ Qty. .... $______

Labor @ 3¢ per copy X______ Qty. .... $______

Periodic advertisement X______ Qty. .... $______
(See Table III)

Press release preparation & mailing @ $150/release X______ Qty. .... $______

Total, Advance Program .... $ 4,000

13.4 Final Program

Artwork & Typesetting – Includes handling charge
(No. of pages _____ X $350) .... $______

Printing: ______ Qty. .... $______
See Table I

Mailing (See Table II)
(The final program is usually distributed only at the conference)
Postage (Bulk) @ 9.0¢ X______ Qty. .... $______

Postage (First Class) (22¢, 4 pages; 37¢ m.t 4; + 3¢/envelope) X______ Qty. .... $______

Mailing Labels (see Table II) @ 4.5¢ X______ Qty. .... $______

Labor @ 3¢ per copy X______ Qty. .... $______

Periodic advertisement .... $______
See Table III
(The final program is usually distributed only at the conference)

Press release preparation & mailing @ $150/release X______ Qty. .... $______

Total, Final Program .... $ 0

Engineering Academic Outreach

Budget Detail

CONFERENCE NAME: **IEEE/CS** Contact : **R.P. Cook 982-2215, 4-7605**

Seventh IEEE Workshop on Real-Time Operating Systems and Software
Estimated and Actual Expenses
May 9-11, 1990

Engineering Academic Outreach
Report 8/1/90

INCOME:

REGISTRATION FEES	Estimated		Actual	
60 Members	@$125	$7500	30 @ $125	$3750
5 Non-Members	@$150	750	4 @ $150	600
0 Students	@		27 @ $ 50	1350
0 One-Day	@	0		
7 Late-Pay Members	@145	1,015	19 @ $145	2755
3 Late-Pay Non-Members	@175	525	7 @ $180	1260
87 Total Paid:				
Total Registration Income		**$9,790**		**$9715**

SOCIAL FUNCTIONS
___0__ Banquet Tickets @ Included fee
OTHER
___0__ Proceedings @ Included fee
___0__ Workshop Participants @ None
___0__ Page Charges @ None

OTHER SOURCES:

	Estimated	Actual
Grant from Office of Naval Research	$10,000	$10,000
Total Income	**$19,790**	**$19,715**

Recap of Participants

	EST	ACT
Members	67	49
Non-Members	8	11
Students	0	27 (10 UVA)
		87 paid participants

Complimentary 4 (Cook, Adrion, Kuhn, Segal)
Attended-Fees not collected 3 (Bunnell, Singh, Zedan)
Conference Workers 2 (Sullivan, Dean)

96
+2 Registered, didn't attend, didn't pay
(Jim Smith and Ralph Wachter)

98 registered

MEETING EXPENSES:

	Estimated	Actual	
PROMOTION			
Honorariums 1 x $1000	$1,000		0
Conference Proceedings,	1,071		
printing and binding			
Printing Services		$434.40	
Academic Outreach		636.40	
			$1071.00
Typing: gratis Cook	0		
Mailing, Fax, Federal Express			33.52
**Call for Papers: gratis Cook	0		
**Final Program: gratis Cook			
***Advertising:	4,000	50.00	1,250
Total Promotion:	**$6,071**		**$2,354.52**
MEETING FUNCTIONS			
Meeting Facilities (if less than 90 room nights	$212		Complimentary 0
Audio-Visual Equipment:			
2 -8x10 screens @ $35x 2 days	70		70
5 Tabletop Mikes @ $8 each x 2 days			80
1 Lavalier Mike @ $20 x 2 days			40
1 Laser Pointer @ $25x2 days			50
Plus 4.5% Tax			250.80
Telephone/Registration Desk			26.12
Transportation			None
Parking			Complimentary
Total Meeting Functions:	**$282**		**$276.92**
COMMITTEE EXPENSE: One trip to Dulles	**1000**		0
CREDIT CARD PROCESSING FEE		$5445@5 %	**$272.25**
ADMINISTRATIVE COSTS:			
Clerical @ !9.70/hour			
Type Brochures	0		
Type Papers	0		
Type Badges (Materials included) 9 hrs.	174		61.80
Registration/Mail/Telephone 24 hrs.	473		717.42
Registration/Desk-First day only, 8 hrs.	158	4 hr.	78.80
Conference Director-gratis		70 hr.	0
Attendees List 8 hrs.	158	1.5 hr.	29.70
Fiscal Administration/Deposits/weekly reports			75.30
Graphics @ $32.70 hr.			
Logo Cover			included above
Overheads			
Total Administrative Costs	**$963**		**$963**
TOTAL MEETING EXPENSES	**$8,316**		**$3,866.69**
Contingency -15% of Total Expense (Minimum $1000)	1,000		0

		Estimated	Actual	
SOCIAL FUNCTION EXPENSES				
Reception: $21.35 x 75 people		1,600	$20.25x95x1	$1,923.02
Banquet: None		0		
Speaker's Breakfast $11.50 x 20 people x 2 day		460	$22.32x25x2	558.03
Lunch	$12.50 x 75 people x2 days	1,875	$15.59x95x2	2,961.62
Breaks: Morning	$6.50 x 75 people x 2days=975		$ 7.52x95x2=1429.42	
Afternoon	$ 3.75 x 75 people x2 days=563		$ 3.58x95x2= 680.55	
	1538=$5.13			
= 4 breaks x $5.25 per person		1,575	$5.55x95x4=	2,109.97
Total Food and Functions (includes 16% gratuity, 7.5% tax on meals and gratuity plus estimated 4% seasonal inflation factor)		**$,5510**		**$7,552.64**
TOTAL EXPENSE		**$15,991**	**$80x95**	**$11,419.33**
14% Fee Paid to IEEE		**1,165**		**265.00**
Purchase Order to IEEE/CS for Newsletter				**8,030.34**
				$19,715.00
TOTAL INCOME				
IEEE Registration Income		**$9,790**		**9,715.00**
ONR Matching Funds				**$10,715.00**
TOTAL INCOME				**$19,715.00**
PLANNED SURPLUS (RETURN TO IEEE)		$3,799		0

***School (Sullivan)**
****Department (Cook)**
*****ONR**

Attendee List for the 7th IEEE Real-Time Workshop, May 10-11, 1990

W. Richards Adrion
University of Massachusetts

William D. Allen
North Carolina State University
6009 Oxford Green Drive
Apex, NC 27502
919-737-2336

Noritake Ashida
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-7673
N.Ashida@k.gp.cs.cmu.edu

Theodore P. Baker
Florida State University
207A Love Building
Tallahassee, FL 32306
904-644 5452
baker@nu.cs.fsu.edu

Tom Bihari
Adaptive Machine Technologies
1218 Kinnear Road
Columbus, OH 43212
614-486-7741
amt@eagle.eng.ohio-state.edu

Scott Breach
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-7656
seb@k.gp.cs.cmu.edu

Dale Brouhard
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92157
619-553-4132

Mitchell Bunnell
Lynx Real-Time Systems, Inc.
550 Division Street
Campbell, CA 95008
408-370-2233

Anthony Burrell
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-982-2296

Scott J. Carter
McDonnell Douglas Electronic Systems Co.
Dept. 233, M/S 28-1, 5301 Bolsa Ave
Huntington Beach, CA 92647
714-896-3077

Shi-Chin Chiang
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-979-1679
scc5k@virginia.edu

Catherine E. Chronaki
University of Rochester
Computer Science Department
Rochester, NY 14627
716-275-7230
chronaki@cs.rochester.edu

Wesley W. Chu
University of California
Computer Science Department
3731 Boelter Hall
Los Angeles, CA 90024
213-825-2047
wwc@cs.ucla.edu

Cristian Constantinescu
Polytechnic Institute of Bucharest
Spl. Independentei 313
Dept. of Control & Computers
Bucharest 77206 Romania
718-786-1221

Robert P. Cook
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-982-2215

B. Dasarathy
Concurrent Computer Corporation
1 Technology Way
Westford, MA 02159
908-392-2639
das@westford.ccvr.com.

Susan B. Davidson
University of Pennsylvania
Department of Computer and Info Science
Philadelphia, PA 19104
215-898-3490
susan@central.cis.upenn.edu

M. Donner
IBM T. J. Watson Research Center

Terry H. Ess
PA Consulting
279 Princeton Road
Hightstown, NJ 08520

Stuart Faulk
Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 22070
703-742-7117

Attendee List for the 7th IEEE Real-Time Workshop, May 10-11, 1990

Robert J. Fornaro
North Carolina State University
Computer Science Department
PO Box 8206
Raleigh, NC 27695
919-737-7848
fornaro@cscadm.ncsu.edu

John Fray
Naval Research Laboratories
Code 8333
4555 Overlook Avenue SW
Washington, DC 20375-5000
202-767-5757

Ahmed M. Gheith
Georgia Institute of Technology
School of Information and
Computer Science
Atlanta, GA 30332
404-894-3982
gheith@cs.gatech.edu

Prabha Gopinath
Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510
914-945-6539
psg@philabs.philips.com

Karen Gordon
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311
703-845-3591
gordon@ida.org

Andrew Grimshaw
University of Virginia
Thornton Hall
Charlottesville, VA 22903
804-982-2204
grimshaw@cs.virginia.edu

Rajiv Gupta
Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510
914-945-6448
gupta@philabs.philips.com

Karen Hargrove
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
206-882-8080
nathanm

Walter Heimeroinger
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN
612-782-7319
heimerdinger@src.honeywell.com

Constance Heitmeyer
Naval Research Lab
Code 5534
Washington, DC 20375
202-767-3596
Heitmeyer@itd.nrl.navy.mil

Ralf Herrtwich
International Computer Science
Institute Berkeley
1947 Center Street, Suite 600
Berkeley, CA 94704
415-643-9153
rgh@icsi.berkeley.edu

Lifeng Hsu
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-982-2291
llh4h.cs.virginia

Kimlan Huynh
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92157
619-553-6367

Farnam Jahanian
IBM Research
PO Box 704
Yorktown Heights, NY 10598
914-784-7498
FARNAM@ibm.com

David Jameson

Kevin Jeffay
University of North Carolina at
Chapel Hill
Dept. of Computer Science
Chapel Hill, NC 27599-3175
919-962-1938
jeffay@cs.unc.edu

Russell Johnston
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92157
619-553-4096

K. H. Kim
University of California
Computer Engineering Program
Dept. of Electrical Engineering
Irvine, CA 92717
714-856-5542
KANE@Ics.Uci.Edu

Young-Kuk Kim
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-979-1051
ykim@virginia.edu

Nobuyoshi Kimura
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-7673
N. Kimura@k.gp.cs.cmu.edu

Robert B. King
University of Pennsylvania
Department of Computer and Info Science
Philadelphia, PA 19104
215-898-0350
king@grasp.cis.upenn.edu

Daniel L. Kiskis
University of Michigan
RTCL 1301 Beal Avenue
Ann Arbor, MI 48109-2122
313-763-6131

R. Kuhn

Bruce Labaw
Naval Research Laboratory
Code 5505
Washington, DC 20375
202-767-3249
labaw@itd.nrl.navy.mil

Thomas F. Lawrence
Rome Air Development Center
39 Proctor Boulevard
Utica, NY 13501
315-330-2158
Lawrence@TOPS20.RADC.AF.MFL

Thomas LeBlanc
University of Rochester
Computer Science Department
Rochester, NY 14620
716-275-5426
leblanc@cs.rochester.edu

Insup Lee
University of Pennsylvania
Department of Computer and Info Science
Philadelphia, PA 19104

Juhnyoung Lee
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-982-2296
j12q@virginia.edu

John P. Lehoczky
Carnegie Mellon University
Department of Statistics
Pittsburgh, PA 15213
412-268-8725
jpl@k.gp.cs.cmu.edu

Kwei-Jay Lin
University of Illinois
1304 W. Springfield
Urbana, IL 61801
217-333-1424
klin@cs.uiuc.edu

Jane W. S. Liu
University of Illinois
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL
217-333-0135
janelui@cs.ulvc.edu

C . Douglass Locke
IBM Systems Integration Division
6600 Rockledge Drive
Bethesda, MD 20817
301-493-1496
cdl@cs.cmu.edu

Evangelos P. Markatos
University of Rochester
Computer Science Department
Rochester, NY 14627
716-275-5414
markatos@cs.rochester.edu

Clifford W. Mercer
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-8744
cwm@cs.cmu.edu

Al Mok
University of Texas
Department of Computer Science
TAY 3-180C
Austin, TX 78712
512-471-4542
mok@cs.utexas.edu

Lory Molesky
University of Massachusetts
Dept. of Computer and Information Science
Amherst, MA 01003
413-545-4822
lory@cs.umass.edu

William Moran

Daniel Mosse'
University of Maryland
Computer Science Department
College Park, MD 20742
301-454-1516
mosse@cs.umd.edu

Nathan Myhrvold
Microsoft Corporation
One Microsoft Way
Building 1/1112
Redmond, WA 98052-6399
206-882-8080
nathanm

Vivek Nirkhe
University of Maryland
Computer Science Department
A. V. Williams Bldg.
College Park, MD 20742
301-454-6153
vivek@amazon.cs.umd.edu

Gultekin Ozsoyoglu
Case Western Reserve University
Department of Computer Engineering and Science
Cleveland, OH 44106
216-368-5029
tekin@alptra.ces.cwcu.edu

Marc Poris
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-295-4475
msp3k@uvacs.cs.virginia.edu

Calton Pu

Peter P. Puschner
Technical University Vienna
Treitlstr. 3/182/1
A-1100 Vienna, Austria
(+43) 222 58801 8169
peter%vmars@relay.EU.net

Ragunathan Rajkumar
IBM Watson Research Center
PO Box 704
Yorktown Heights, NY 10598
914-784-7931
vajkumr@IBM.com

Krithi Ramamritham
University of Massachusetts
COINS Dept., A305 LGRC
Amherst, MA 01003
413-545-0196
Krithi@nirvan. cs.umass.edu

Prithvi Narayan Rao
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-7668
p.rao@k.gp.cs.cmu.edu

Jim Ready
Ready Systems

Ambar Sarkar
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-979-8796
as6n@virginia.edu

Eric Schorman
Motorola, Inc.
PO Box 2931
Fort Worth, TX 76113
817-232-6098

Karsten Schwan
Georgia Institute of Technology
ICS Department
AECAL Building
Atlanta, GA 30332-0280
404-894-2589
schwan@gatech

Chia Shen
University of Massachusetts
Dept. of Computer and Information Science
Amherst, MA 01003
413-545-4753

Terry Shepard
Royal Military College
Department of Electrical and
Computer Engineering
Kingston, Ontario, Canada K7K 5L0
613-545-2363
Shepard @ rucis.queensu.ca

Kang G. Shin
University of Michigan
Dept. of Electrical Engineering and Computer
Science
Ann Arbor, MI 48109-2122
313-763-0391
kgshin@dip.eecs.umich.edu

Inder M. Singh
Lynx Real-Time Systems, Inc.
550 Division Street
Campbell, CA 95008
408-370-2233

Jim Smith
Office of Naval Research

Sang H. Son
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22903
804-982-2205
Son @ CS. Virginia.edu

W. H. Spotz

John A. Stankovic
University of Massachusetts
COINS Dept., A305 LGRC
Amherst, MA 01003
413-545-0720
stankovic@cs.umass.edu

Tim Strayer
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-296-9897
wts4x@virginia.edu

Jay Strosnider

Jay K. Strosnider
Carnegie Mellon University
ECE Department
5000 Forbes Avenue
Pittsburgh, PA 15213
412-268-6927
jks.cetus.ece.cmu.edu

Johan Sunter
University of Twente
Thomas de Reyserstraat 215
7545 BG Emchade
Holland
053-893885

K. C. Tai
National Science Foundation/
North Carolina State University

Andre' M. Van Tilborg
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5000
202-696-4302
avantil@uswc-wo.arpa

Hideyuki Tokuda
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
412-268-7672
hxt@cs.cmu.edu

Ralph Wachter
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5000
202-696-4302
wachter@itd.nrl.navy,mil

Prasad Wagle
University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
804-982-2298
psw4j@uvacs.cs.virginia.edu

Tom Walliser
Adaptive Machine Technologies
1218 Kinnear Road
Columbus, OH 43212
614-486-7741

Pat Watson
IBM Federal Systems Division
250/060 9500 Godwin Drive
Manassas, VA 22110
703-367-4536
Watson@k.gp.cs.cmu.edu

Horst Wedde
Wayne State University

Victor Wolfe
University of Pennsylvania
Department of Computer Science
Philadelphia, PA 19104
215-243-7009
wolfe@grasp.cis.upenn.edu

Mark D. Wood
Cornell University
653 Coddington Road
Ithaca, NY 14850
607-255-1149
wood@cs.cornell.edu

Albert Yu

H. Zedan
University of York
Computer Science Department
York Y01 5DD England
United Kingdom
Zedan@UK.ac.york.minster

F C

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

UNIVERSITY OF VIRGINIA

OFFICE OF THE DEAN
THORNTON HALL

NOV 14 1989

CORRESPONDENT'S PHONE
(804) 924-

Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5000

Attention: Dr. Andre van Tilborg, Director
Computer Science Division, Code 1133

Dear Dr. Tilborg:

Enclosed for your review and evaluation are six (6) copies of our detailed budget for the proposal entitled "Seventh IEEE Workshop on Real-Time Operating Systems and Software."

If we can be of further assistance, please contact Mr. Gilbert Hay, Director of Administration, at 804-924-3310.

Sincerely,

E. A. Starke, Jr.
Earnest Oglesby Professor
and Dean

EASjr:ph

Enclosures: (6) Detailed Budget for SEAS Proposal No. CS-DOD/ONR-4511-90

c: Mr. Michael McCracken, ONR
Dr. R. P. Cook, CS
Dr. A. K. Jones, CS
Mr. D. W. Jennings, UVa.
Ms. R. A. Nixon, UVa.
Ms. F. B. Cline, UVa.
Ms. D. E. Van, UVa.

CHARLOTTESVILLE, VIRGINIA 22901

GRANT CHANGE REQUEST
Federal Demonstration Project

Table 3

To: OFFICE OF SPONSORED PROGRAMS - University of Virginia

A. Program Director: R. P. Cook Date: November 1989

Department: Computer Science Account Code:

Sponsor: DOD/ONR Grant No.:

Budget Period: 5/1/90 to 5/31/90

Change No. ______ to this account CS-DOD/ONR-4511-90

B. Sponsor Approval Requested For: (Attach Letter to Sponsor)

___Scope or Objective Changes ___Change or Absence of PI ___*Establish Relatedness

___**Foreign Travel ___**No Cost Extention [✓ X] Other (specify) Define budget

* Complete Section D. and Relatedness Form (Dean's signature required only if more than one department involved).

**See reverse for agencies requiring prior approval.

C. Institutional Approval Requested For: (See reverse for requirements)

___*Preaward Costs ___No Cost Extension ___Rebudgeting from Direct Costs to Indirect Costs and Vice Versa

* Department chair agrees to cover all expenditures not reimbursed by the funding agency.

D. Establish Project Relatedness with the Following Grants:

Principal Investigator	Grant No.	OSP Acct. No.

E. Explanation/Justification: (Briefly cite scientific, technical or administrative reasons(s) for this request.)

Sponsor requested detailed budget.

F. Certification/Approvals:

The scientific and technical propriety of this request has been reviewed and approved. This request is consistent with the scope and objectives of the approved project.

Principal Investigator R. P. Cook Date | Department Chairperson A. K. Jones Date

Research Administrator G. Hay Date | Dean (If required per Section B above) Date

Institutional Official D. W. Jennings Date

FORM SP-23B (10/1/88)

OSP REQUIRES ONE ORIGINAL COPY

SEVENTH IEEE WORKSHOP ON REAL-TIME
OPERATING SYSTEMS AND SOFTWARE

PROPOSAL NO. CS-DOD/ONR-4511-90

Submitted by R. P. Cook
Associated Professor of Computer Science
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903-2442

DETAILED BUDGET

May 1990

Committee Expenses		
Travel - program arrangements, selection of papers, speakers		$1,000
Registration Costs		
Mail/telephone registration	$473	
On site registration, 1 day	158	
		631
Administrative Costs		
Speaker-honorarium	$1,000	
Audio-visual equipment use	70	
Conference proceedings booklet copying costs	1,071	
Meeting space rental	212	
Badges, reports, deposits, etc.	332	
		2,685
Food and Functions		
Meeting for informal discussions	1,600	
Speaker's breakfast/lunch/break inc. gratuities, taxes, fees	3,910	
		5,510
Contingency Funds		174
TOTAL		*$10,000

*Any excess funds will be used for special issue of the IEEE Real-Time Technical Committee Newsletter.

OFFICE OF SPONSORED PROGRAMS (OSP)
NOTICE OF AWARD

CS-DOD/ONR - 4511-90

University Account No. 5-25442 Date of this Notice: JANUARY 3, 1990

The University of Virginia has entered into an agreement as herein described. Initiation of the expenditure of funds is authorized in accordance with the terms and conditions set forth in the attached sponsor's award document and/or policy statement.

Program No. 10.000
BRS Subcode:

[illegible]
Office of Sponsored Programs

PROGRAM DIRECTOR: COOK RP UNIT: Computer Science

UNIVERSITY ADMINISTRATOR: MS. CLINE TELEPHONE EXT. (804) 924-3931

PROJECT TITLE: WORKSHOP ON REALTIME OPERATING SYSTEMS & SOFTWARE

SPONSOR: ONR (0500) GRANT/CONTRACT NO. N00014-90-J-1339

CURRENT BUDGET PRD: 01/01/90 TO 12/31/90 PROJECT PRD: 01/01/90 TO 12/31/90

AWARD: $10,000.00 DIRECT COSTS: $10,000.00 INDIRECT COSTS: $0.00

PROGRAM TYPE: Service PROGRAM STATUS: New OLD ACCOUNT NO. 0-00000

REPORTING REQUIREMENTS		FREQUENCY	FINAL REPORT DUE	REMARKS
x	Technical Reports:	Annually	02/28/91	copy of report or letter to OSP
x	Fiscal Reports:	Annually	03/30/91	
x	Patent Reports:	Annually	02/28/91	
___	Equipment:			

RENEWAL OR CONTINUATION APPLICATION DUE:

PROPERTY AND EQUIPMENT CONDITIONS:

___ Sponsor or University approval required for specific items costing in excess of
___ Sponsor or University approval required for general purpose items costing in excess of

OTHER SPECIFIC CONDITIONS:

___ Requires Sponsor approval for Foreign Travel.
___ Requires Sponsor approval for any deviation from approved budget.
___ Requires cost sharing.
x Indirect Cost Rate 0 % of Salary & Wages MTDC TDC x Other
___ Computer Funds:

Comments: IDC not applicable.Monthly billings for reimbursement .
3 copies of proceedings required within 60 days of completion.
Prior approval required for travel from Communist Bloc countries.

LEDGER POSTED

OSP Proposal No. P90-15004 OSP Award No. A90-1

§§§§ORIGINAL BUDGET¶¶¶¶

ACCOUNT: 5-25442 PI: COOK RP DATE: 01/03/90
PROPOSAL: 15004

LEVEL BUDGET CATEGORIES		CURRENT BUDGET	NET CHANGE
2 - PERSONAL SERVICES	**	$	$
2 - OTHER THAN PERSONAL SERVICES	***	10,000.00	
3-Fringe Benefits			
3-Consultant Services			
3-Technical Services			
4-Computing			
3-Travel			
4-Foreign Travel			
3-Health Care Services			
3-Other Contr. Services			
4-Publications			
3-Supplies			
3-Equipment			
3-Current Charges			
2 - SCHOL. & FELLOWSHIPS	***		
3-Trainee Travel			
3-Trainee Stipends			
3-Tuition & Fees			
3-Other Payments			
2 - RENOVATIONS	***		
TOTAL DIRECT COSTS		10,000.00	0.00
TOTAL INDIRECT COSTS			
TOTAL RESTRICTED			
TOTAL BUDGET		10,000.00	0.00
LESS CARRYOVER		0.00	
NET AWARD		10,000.00	

* The number preceding the budget categories indicates the budgeting level.
** Personal Services may ONLY be budgeted at level 2.
*** NO amount may be budgeted at level 2 if level 3 and 4 budgeting is required.

BUDGETING GUIDELINES

1. Level 2 budgeting is recommended whenever sponsor requirements permit (i.e. non-Federal unrestricted gifts and grants).
2. Level 3 will be required for most all Federal grants and contracts because of standard budget restrictions and format
3. Level 4 budgeting should only be used when required by the Research Administrator or sponsor restrictions (i.e. foreign travel, computing, publications).
4. A positive category balance is mandatory for budget reallocations.

DEPARTMENT OF THE NAVY
OFFICE OF NAVAL RESEARCH, CODE 1513:RAR
800 NORTH QUINCY STREET

ARLINGTON, VIRGINIA 22217-5000

GRANT NO: N00014-90-J-1339
FORMERLY GRANT NO: N00014-90-G-0339
R&T PROJECT: 4331778---01
ACO CODE: N66002
CAGE CODE: 1B752
DISBURSING CODE: N00179

DUPLICATE ORIGINAL

SYMPOSIUM GRANT

GRANTEE: University of Virginia
Carruthers Hall, P.O. Box 9003
Charlottesville, VA 22906

APPROPRIATION: AA 1701319.W1AE

Object Class:	000
Unit Ident Code:	RA434
Suballotment:	0
Auth. Acct. No:	068342
Transaction Type:	2B
Prop. Acct. Act.:	000000
Cost Code:	015080000100
Amount:	$10,000.00
FRC:	4331

R&T Project Code: 4331778---01, Dated: 08 NOV 1989

TOTAL GRANT AMOUNT: $10,000.00

AUTHORITY: 10 USC 2358 as amended, and 31 USC 6304.

GRANT PURPOSE: The Purpose of this Grant is to provide partial funding to support (u)Seventh IEEE Workshop on Real-Time.

The conduct of the workshop, the personnel and effort and the use of funds for direct and indirect expenses shall generally be as set forth in the Grantee's proposal entitled "Seventh IEEE Workshop on Real-Time Operating Systems and Software", dated 17 OCT 1989 which proposal is incorporated herein by reference. The Grantee agrees to obtain concurrence of the Grantor for any desired deviation from the proposal.

PERIOD: The Grant is for the period 01 JAN 1990 through 31 DEC 1990.

PRINCIPAL INVESTIGATOR: The Principal Investigator, Professor Robert P. Cook shall be continuously responsible for the conduct of the project. The Grantee agrees to obtain approval of the Grantor before changing the Principal Investigator.

SCIENTIFIC OFFICER: The Scientific Officer representing the United States Government under this Grant is Gary M. Koob, Code 1133, Office of Naval Research, 800 North Quincy Street, Arlington, Virginia 22217-5000.

GRANTS ADMINISTRATOR: The Grants Administrator for this Grant is:
Office of Naval Research
Resident Representative N66002
Administrative Contracting Officer

National Academy of Sciences
2135 Wisconsin Ave., NW Suite 102
Washington, DC 20007-3259

PAYMENTS: Upon submission of invoices by the Grantee in accordance with the provisions of this Grant, the amount specified herein shall be paid as set forth in Attachment Number 1. Invoices hereunder shall be submitted by the Grantee in sextuplicate to the Grants Administrator for certification and transmittal to the Navy Regional Finance Center, Crystal Mall #3, Rm. 260 Attn: Code 431, Washington, D.C. 20371-5400, where payment will be made.

The Grantee is participating in the cost of this effort.

PERFORMANCE REPORTS AND/OR PROCEEDINGS: (a) The Grantee shall submit to the attached distribution list the following documents within 60 days after the end date of this grant:

3 copies of the Proceedings.

(b) The Grantee shall include a complete "Document Control Data - R&D" form (DD Form 1473) as the last page of each copy of every scientific and technical report prepared under this Grant. The form contains instructions for preparation. The cognizant Government Grant Administrator will provide assistance to the Grantee in obtaining the required forms. Administrative type reports, managerial (status) reports, and reprints submitted as technical reports are excluded from this requirement.

PATENTS AND COPYRIGHTS: (a) With respect to patents and other rights arising out of inventions, improvements or discoveries conceived or first actually reduced to practice during the effort, Grantee shall (1) give and hereby does grant to the United States Government an irrevocable, non-exclusive, non-transferable royalty-free license to practice or have practiced for its benefit, each invention (whether or not patentable) throughout the world, (2) advise Grantor of the filing of each patent application in any country and furnish a copy thereof to Grantor, (3) give Grantor the right to file patent application(s) for any invention on which Grantee does not intend to file, as to which inventions the Government is hereby granted sole and exclusive title, and (4) on request, furnish grantor duly executed instruments fully confirmatory of said license and/or title rights.

(b) The Government shall have the right to publish, translate, reproduce, deliver, and dispose of all data, including reports, drawings, blueprints, and technical information which are delivered to the Government under this Grant, and to authorize others to do so. With respect to data which are not originated during the effort Grantee shall give a similar license but only to the extent that Grantee and those in privity with Grantee have the right to give such license without paying compensation to others because of giving the license. At the time of giving or reporting any such data, Grantee shall make all reasonable effort to advise Grantor (1) of all invasions of the right of privacy contained therein and, (2) of all portions of such data copied from work not composed or produced during the effort and not licensed under this provision.

(c) Notwithstanding the provision of the preceding paragraph, the Grantee and the Government may agree that specifically designated data shall not be published for sale by the Government, nor shall the Government authorize others to do so when such data are published by the Grantee, and shall so refrain so long as the data are protected by copyright.

RESTRICTIONS ON PRINTING: Unless otherwise authorized in writing by the Grants Officer, reports submitted hereunder shall be reproduced only by duplicating processes and shall not exceed 5,000 single page reports or a total of 25,000 pages of a multiple-page report. To satisfy the requirement of the Defense Technical Information Center the copy of the technical report submitted to the Defense Technical Information Center must be black typing or reproduction of black on white paper or suitable for reproduction by photographic techniques. Reprints of published technical articles are not within the scope of this paragraph.

PUBLICATIONS:
1. Any publication resulting from work under this Grant shall contain the following on the title page or on the page immediately following the title page:

> This work relates to Department of Navy Grant N00014-90-J-1339 issued by the Office of Naval Research. The United States Government has a royalty-free license throughout the world in all copyrightable material contained herein.

2. Any transfer of copyright ownership in such publication will provide that the transfer of copyright ownership is subject to the U.S. Government's royalty-free license throughout the world in all copyrightable material contained in the publication.

CIVIL RIGHTS ACT: This Grant is subject to the compliance requirements of the "Civil Rights Act of 1964," 78 Stat. 241 (Public Law 88-352) relating tc nondiscrimination in Federally assisted programs. The Grantee has signed an Assurance Compliance with the nondiscriminatory provisions of the Act.

4

FINANCIAL RECORDS AND REPORTS: The Grantee shall maintain adequate records to account for the expenditures made under this Grant and, when applicable to this document the actual amount of cost participation. Upon completion or revocation of this Grant, whichever occurs earlier, the Grantee shall furnish to the Grants Administrator, a Financial Status Report in accordance with OMB Circular A-110, Attachment G, Exhibit 1, showing a breakdown of expenditures made in performance of this Grant. Such financial statement may be on a cash or accrual basis depending upon the Grantee's accounting system. Such financial statement may be in the same detail as contained in the Grantee's approved budget for this Grant and shall be submitted no later than 90 days after the end of the annual reporting period or completion or revocation of the Grant. The Grantee's financial records are subject to audit by the Government when desired by the Grantor.

UNEXPENDED FUNDS AND EARNED INTEREST: After the end of the Grant period, any uncommitted funds and any interest earned by Grant funds on deposit shall be returned to the Office of Naval Research by check made payable to "Office of Naval Research."

RESTRICTION ON TRAVEL: The Grantee must obtain prior written approval from the Grants Officer, before funds provided under this Grant may be expended to provide for travel for persons from Communist Bloc countries.

TRAVEL BY GOVERNMENT EMPLOYEES: Funding of travel by employees of the U. S. Government with funds provided under this Grant is prohibited.

REVOCATION: This Grant may be revoked in whole or in part by the Grants Officer after consultation and agreement with the Grantee, provided that such revocation shall not affect any commitment which, in the judgment of the Grants Officer and the Grantee, has become firm prior to the effective date of the revocation; and funds not committed by the Grantee prior to the revocation shall be returned to the Office of Naval Research.

UNITED STATES OF AMERICA

FOR THE OFFICE OF NAVAL RESEARCH

BY: [illegible]

(GRANTS OFFICER) ROYAL A. RUCKER

DEC 1 3 1989

(DATE)

GRANT NO: N00014-90-J-1339

ATTACHMENT NUMBER 2

Distribution List for Reports

ADDRESSEE	NUMBER OF COPIES
Scientific Officer Code: 1133 Gary M. Koob Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217-5000	3 copies of proceedings
Grant Administrator Office of Naval Research Resident Representative N66002 Administrative Contracting Officer National Academy of Sciences 2135 Wisconsin Ave., NW Suite 102 Washington, DC 20007-3259	1 copy of proceedings
Defense Technical Information Center Building 5, Cameron Station Alexandria, Virginia 22314	1 copy of proceedings

October 6, 1989

Ms. Maggie Johnson
IEEE Computer Society
1730 Massachusetts Avenue, NW
Washington, DC 20036-1903

Dear Ms. Johnson:

Enclosed please find a completed TMRF for the *Seventh IEEE Workshop on Real-Time Operating Systems and Software* to be held in Charlottesville on May 10-11, 1990.

As you can see from the TMRF, we anticipate a surplus from this workshop, as has been the case with each of the previous six workshops in this series. Note that the US Navy Office of Naval Research will provide a $10,000 grant to the workshop, and Virginia's Center for Innovative Technology has been asked for $2000, reducing the IEEE/CS risk significantly. Also attached is a full-page advertisement which will appear in the IEEE Computer Magazine and in Communications of the ACM. A direct mailing will also be conducted. In addition, the workshop Program Committee will meet at the Dulles International Airport near Washington, DC on March 31, 1990 to select a technical program. As a result, the requested advance of $7000 from IEEE/CS is needed as soon as possible to cover these costs.

Thanks for your help, please call me at 804-924-7605 if I can be of assistance to you.

Sincerely,

Robert P. Cook
Computer Science Department

/ss

Enclosures:	TMRF
	Advertising

c: Dr. K. Lin
Dr. S. Son
Dr. R. Lowry
Ms. S. Sullivan

DISTRIBUTION LIST

1 - 3	Scientific Officer Code: 1133 Gary M. Koob Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5000
4	Grant Administrator Office of Naval Research Resident Representative N66002 Administrative Contracting Officer National Academy of Sciences 2135 Wisconsin Avenue, N. W., Suite 102 Washington, DC 20007-3259
5	Defense Technical Information Center Building 5, Cameron Station Alexandria, VA 22314
6	Dr. Krithi Ramamritham Department of Computer and Information Science Lederle Graduate Research Center University of Massachusetts Amherst, MA 01003
7 - 8	R. P. Cook, CS
9	A. K. Jones, CS
*	E. H. Pancake, Clark Hall
10	S. Sullivan, Academic Outreach
11	SEAS Preaward Administration Files

*Cover letter only

JO#3389:ph